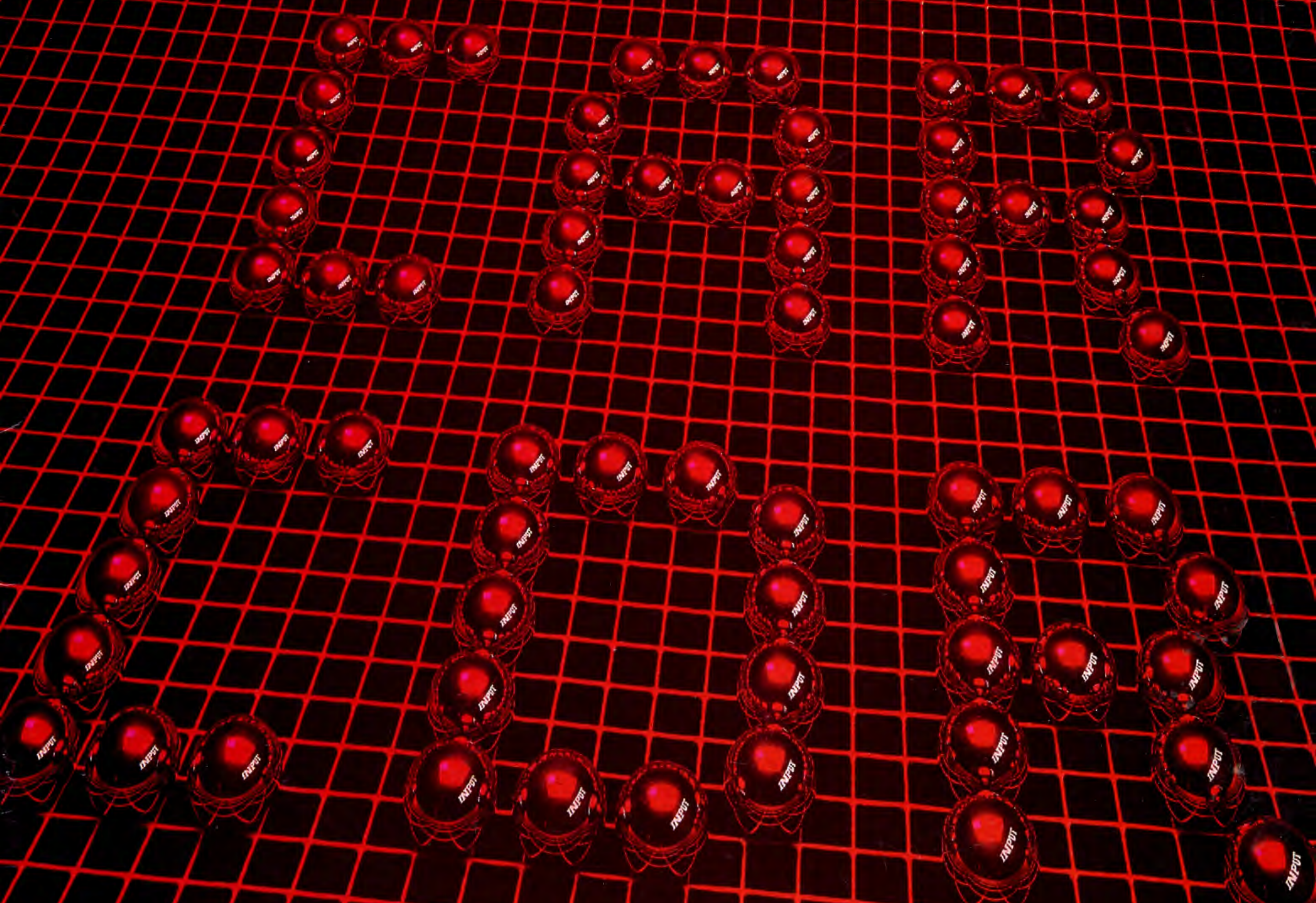
A MARSHALL CAVENDISH 45 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol. 4 No 45

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Graeme Harris. Page 1347, Image Bank/Berry Fallon Design. Pages 1402, 1404, 1406, 1409, Peter Reilly. Pages 1410, 1412, Graeme Harris. Pages 1417, 1418, 1420, 1423, Kevin O'Keefe. Pages 1425, 1426, 1427, 1428, Artist Partners/Stuart Robertson.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Typeset by MS Filmsetting Limited, Frome, Somerset. Printed by Cooper Clegg Web Offset Ltd, Gloucester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +

COMMODORE 64 and 128

ACORN ELECTRON, BBC B and B+

DRAGON 32 and 64

ZX81

VIC 20

TANDY TRS80 COLOUR COMPUTER

MODELS OF IRREGULARITY

- GEOMETRIC FIGURES
- SIZING AN OBJECT
- SELF-SIMILARITY
- FRACTALS AND GRAPHS
- DRAWING DRAGONS

Use your micro to explore the fascinating forms of fractal geometry—the mathematical tool that helps explain the irregularity that shapes the real world

How long is a piece of string? The answer is easy to find out if the string is straight, because all you have to do is to measure it. The same applies even if the string is tangled, except that now you have the rather harder task of measuring an irregular object.

Of course, anyone with any sense would pull the string out straight first, then measure it. But what if you have to measure something which is a complex shape and can't be unravelled?

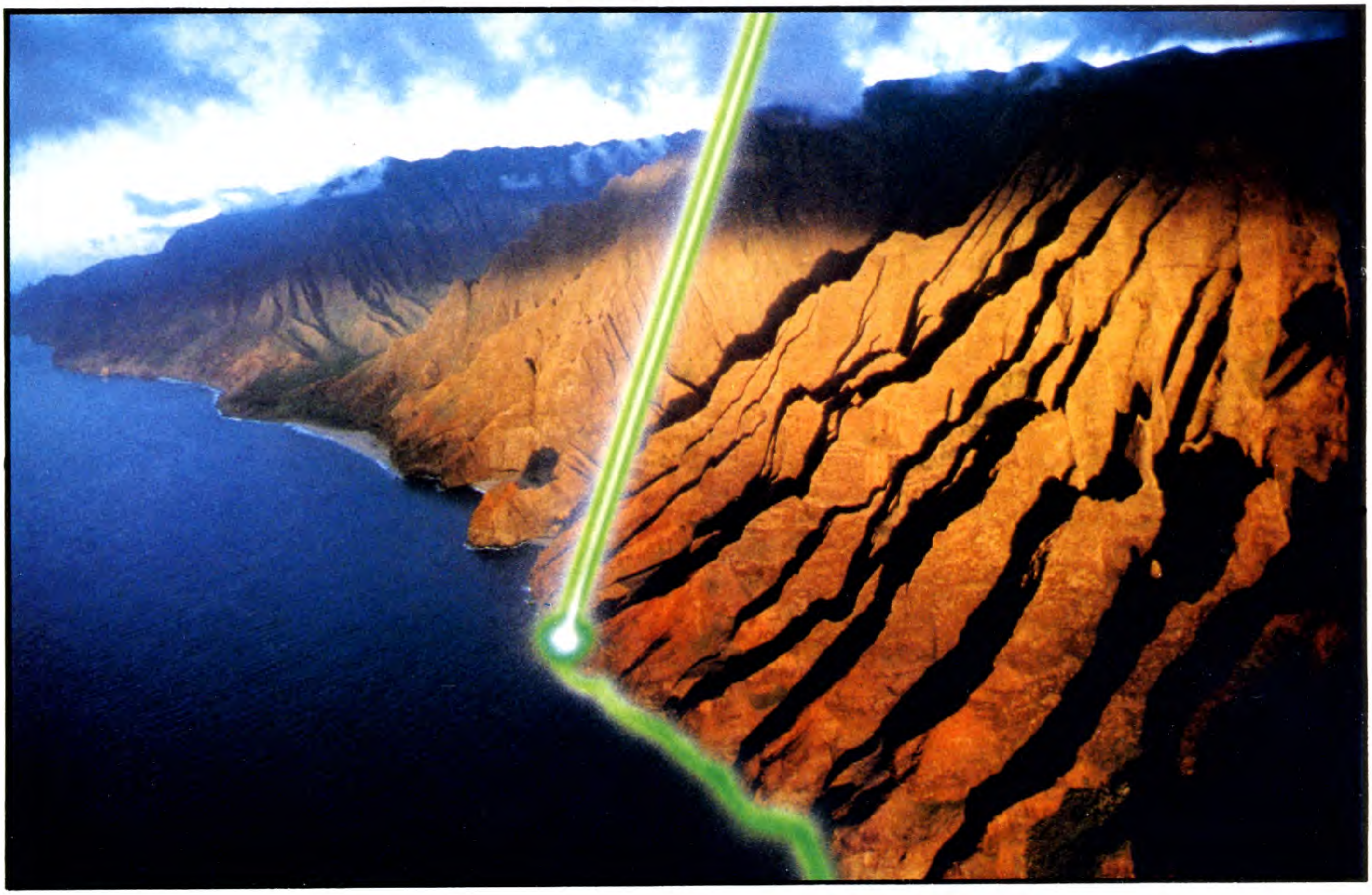
Think of a stretch of rocky coastline, five kilometres long. What picture does this conjure up? Does five kilometres mean the distance as the crow flies, or does it mean the distance you would walk if you actually followed each contour of the shoreline? There is a considerable difference, and it should be obvious that the route which follows every irregularity will be a good deal longer than the straight-line path.

But how much longer? Let's suppose you have a very long, very flexible, tape measure and you decide to measure around every detail. First of all, you have to include all the major irregularities in the coast, like the inlets formed by rivers. But there are smaller irregularities, too, like individual outcrops of rock. As you carry on taking your tape measure around, you can include even smaller details, right down to individual stones, or even grains of sand. And if you were able to look in finer detail still, you would see that the grains of sand themselves have surface irregularities around which to measure.

This might seem an absurd exercise—after all, no-one needs to know distances in such fine detail. But it makes an important point, which is that when you are dealing with something whose outline or surface is not perfectly smooth, its actual measurement depends on the size of your ruler.

Traditional science uses models for curves and surfaces which are assumed to be smooth. And the more they are magnified, the more all of these shapes start to look flat—in the same way that to a rough approximation, the shape of the Earth is a sphere—but so large that to those who live on it, the surface appears flat.

But as the example of the coastline shows,



some objects do not look flat, even when they are under high magnification. And there are many examples in nature of things which possess detailed structure on many different scales. But only recently have scientists and mathematicians identified them as something worthy of study and derived a model for their structure in a new series of geometric figures called *fractals*.

The word fractal comes from the Latin word for irregular. It was coined by a US mathematician, Dr Benoit Mandelbrot, who recognized fractals as a new breed of mathematical object, well suited to modelling irregular, natural objects.

The difference between a fractal and a smooth curve is dramatic in appearance, but it is equally dramatic in theory. The idea of the three dimensions of space is familiar to most people—a surface has only two dimensions, and a line has only one. And to explain some of the physical theories of his day, Albert Einstein suggested a fourth dimension—time. Fractals extend the idea of dimensions in a remarkable way—by requiring fractional, instead of whole number, dimensions.

This is not to say that a fractal requires one-and-a-half or two-and-a-half independent directions—half a direction is meaningless, even to a mathematician. Instead, a new kind of dimension is envisaged, depending on how the fractal behaves under magnification. This new kind of dimension gives the expected answers 3, 2, or 1 for space, a surface, or a curve, but it also makes sense for a fractal.

SIZING AN OBJECT

If you take two pieces of string the same length and place them end to end, you can imagine you have formed a copy of the original piece of string, but twice its size. Starting with a square of paper, however, you would require three copies with the original—four pieces in all—to produce a square of paper twice the size. For a cube of cheese, you would require eight copies. And if four-dimensional cheese existed, you would need 16 identical portions to double the size of one of the pieces.

The numbers 2, 4, 8 and 16 are part of a sequence formed by multiplying 2 by itself several times—2, 2×2, $2\times2\times2$, and $2\times2\times2\times2$. This sequence can also be written 2^1, 2^2, 2^3, 2^4, and so on. And the powers to which 2 is raised—1, 2, 3, 4, and so on—are the dimensions of the object.

A one-dimensional object (the string) has to be multiplied by 2, a two-dimensional object (the paper) by 4, and so on. Now suppose an object could be found that requires *three* identical pieces to double the size of a single piece. This number lies between two (corresponding to dimension 1) and four (dimension 2), so you could say that the dimension of the object lies between 1 and 2.

At first sight, this might appear to be an improbable idea, although the logic is obvious. But it turns out that this theory manages to explain the magnification effect that you discover when trying to measure a fractal.

It is even possible to draw a diagram of something which behaves in this way. The German mathematician von Koch was one of the first to do so when he invented the so-called snowflake curve—which looks like an infinitely crinkled snowflake. Each side of this is made out of four copies of itself, each one third of the size, and the dimension is just over 1.26. And in the natural world, a typical coastline can be viewed as having very similar dimensions.

The fractal curve is defined very precisely, whereas in the natural world there is far more variation, so as in the case of most mathematical models, the artificial situation gives only an approximation to nature. Nevertheless, it provides science's best explanation for many natural structures, ranging from the veins and arteries of the body, the shapes of mountains, the bends of a river, to the bark of a tree.

SELF-SIMILARITY

As you will see if you look back to the examples of the string, the paper and the cheese given above, the object was assembled from smaller copies of itself. A large block of cheese is made of several smaller blocks of cheese the same shape, for example. The object is said to be *self-similar*, in that if you look at any part of it, the object appears to be a smaller version of itself.

Mathematical fractals apply this principle very rigidly, which is what makes them so regular. Natural fractals are not precisely self-similar, so if you magnify a section of a coastline, or a piece of bark, it does not exactly match the structure it was taken from. However, it does look roughly the same, as if it could be a piece of coastline from elsewhere, or bark from another part of the tree. This is called statistical self-similarity and is very common in nature.

FRACTALS AND GRAPHICS

So what does all this have to do with computing? The answer is that as with so many types of mathematical model, fractals can be used to program your computer so that it generates a model of reality, sharing some properties with the real world. Specifically, fractals are finding major applications in computer graphics, where they provide the most practical answer to the problems of generating realistic irregular shapes for things like mountains, seas, and a host of imaginary landscapes.

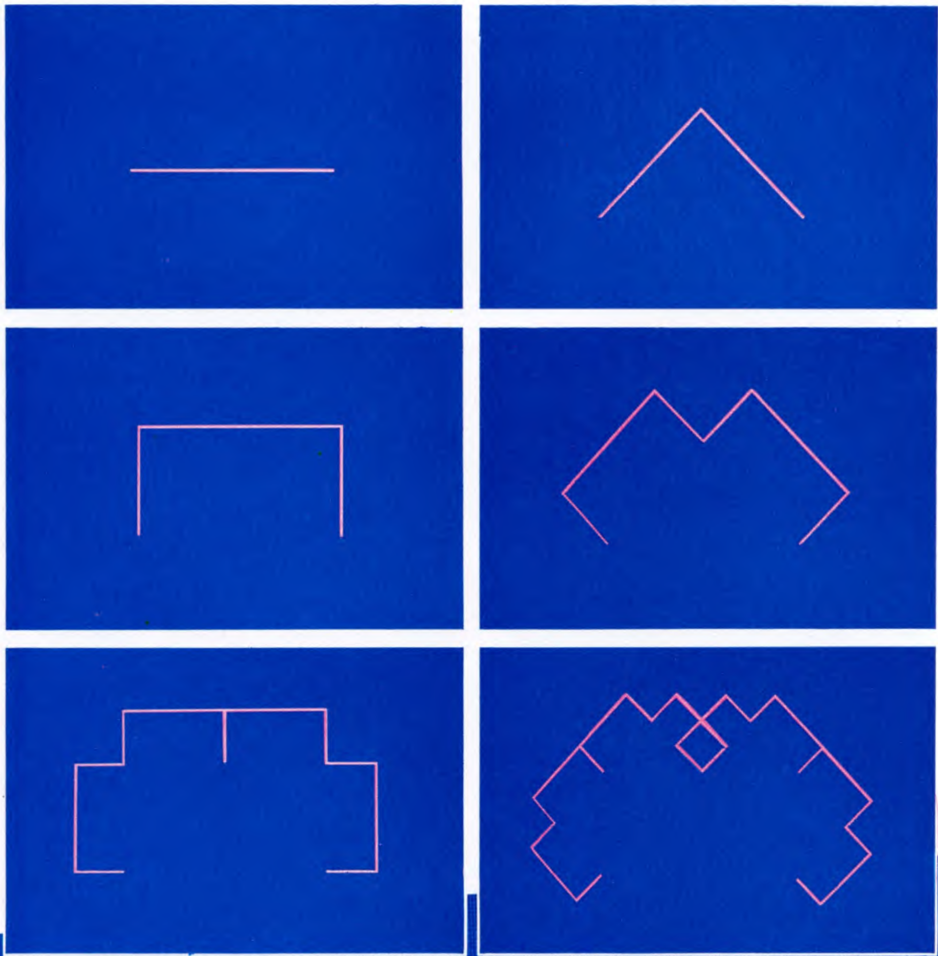
Not only do fractals provide the best mathematical model yet devised, but they are also ideally suited to programming. The principle of self-similarity means that the shape can be built up by generating the same element again and again—a process which lends itself to simple, recursive programs of the type covered in the earlier article on pages 1289 to 1295.

All you need to do to generate a mathematical fractal is to take a simple shape, then repeat this again and again at different scales, gradually adding more and more detail to the drawing. The simplest example of this is just to take a straight line, and apply a rule which says that every straight line must be replaced by a pair of lines at right angles to each other.

If you try drawing this out yourself, you will see that each step adds a new level of detail, as each line in the drawing is itself broken up into two new lines. And gradually, a pattern starts to emerge. The first program gets the computer to demonstrate it for you. Like all fractal processes, the drawing could go on infinitely, adding more and more detail, but this soon would get too fine to display on a television screen. So the program is designed to stop once it has reached a certain level of recursion.

```
1Ø BORDER Ø:BRIGHT 1:PAPER Ø:INK 7:CLS
2Ø LET mn=2
3Ø LET c=PI/18Ø
4Ø LET l=12Ø:LET x=7Ø:LET y=5Ø:LET
   an=PI/2
45 PLOT INVERSE 1;OVER 1;x,y
5Ø GOSUB 1ØØØ
8Ø STOP
1ØØØ LET l=l/1.414
1Ø1Ø IF l<mn THEN LET l=l*1.414:LET
   x=x+(l*SIN (an)):LET y=y-(l*COS
   (an)):DRAW x-PEEK 23677,y-PEEK
   23678:RETURN
1Ø2Ø LET an=an+PI/4:GOSUB 1ØØØ
1Ø3Ø LET an=an-PI/2:GOSUB 1ØØØ
1Ø4Ø LET an=an+PI/4:LET
   l=l*1.414:RETURN
```

```
1Ø PRINT "♥"
2Ø MN=2
3Ø C=π/18Ø
4Ø L=16Ø:X=8Ø:Y=4Ø:AN=π/2:XX=X:
   YY=Y
5Ø HIRES Ø,14:GOSUB 1ØØØ
6Ø GOTO 6Ø
1ØØØ L=L/1.414
```

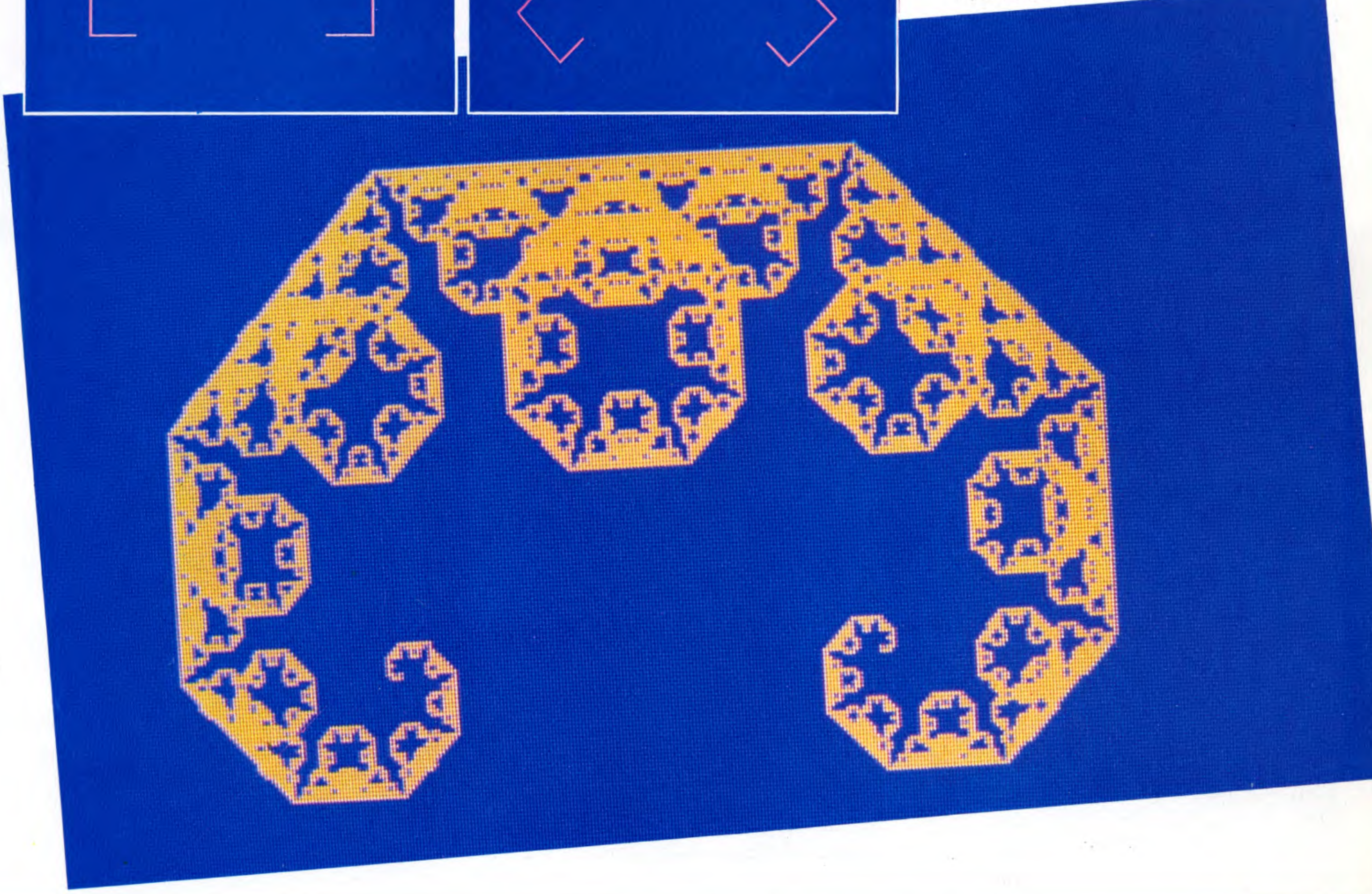


```
1010 IF L>=MN THEN 1020
1012 L=L*1.414:X=X+(L*SIN(AN)):
   Y=Y-(L*COS(AN))
1015 LINE X,Y,XX,YY,1:XX=X:YY=Y:
   RETURN
1020 AN=AN+π/4:GOSUB 1000
1030 AN=AN-π/2:GOSUB 1000
1040 AN=AN+π/4:L=L*1.414:RETURN
```

The program is as for the Commodore 64, except for the following lines:

```
40 L=400:X=300:Y=300:AN=[π]/2:
   POINT 0,X,Y
50 GRAPHIC 2:GOSUB 1000
1000 L=L/1.414:REGION RND(1)*6+2
1015 DRAW 1 TO X,Y:RETURN
```

```
10 MODE1:VDU 23;8202;0;0;0;
20 M%=15
30 VDU19,0,4;0;
40 MOVE 350,260
50 PROCDRAW (600,0)
60 REPEAT UNTIL FALSE
1000 DEF PROCDRAW(L%,A)
1010 IF L%<M% THEN PLOT 1,L%*COS
   A,L%*SIN A:ENDPROC
```



```
1Ø15 L%=L%/1.4142
1Ø2Ø PROCDRAW(L%,A+PI/4)
1Ø3Ø PROCDRAW(L%,A-PI/4)
1Ø4Ø ENDPROC
```

```
1Ø PMODE4,1:PCLS:SCREEN1,1
2Ø MN=2
3Ø C=ATN(1)/45:PI=4*ATN(1)
4Ø L=12Ø:X=7Ø:Y=14Ø:AN=PI/2
45 LINE-(X,Y),PRESET
5Ø GOSUB1ØØØ
6Ø GOTO6Ø
1ØØØ L=L/1.414
1Ø1Ø IF L<MN THEN L=L*1.414:
   X=X+(L*SIN(AN)):Y=Y+(L*COS(AN)):
   LINE-(X,Y),PSET:RETURN
1Ø2Ø AN=AN+PI/4:GOSUB1ØØØ
1Ø3Ø AN=AN-PI/2:GOSUB1ØØØ
1Ø4Ø AN=AN+PI/4:L=L*1.414:RETURN
```

RUN the program and you will see a C-shaped pattern gradually build up on the screen. The program starts by drawing a line, then replaces it with the first pair of lines at right angles. The subroutine between Lines 1ØØØ and 1Ø4Ø, which uses SIN and COS to work out the angles required, is called at Line 5Ø, and it calls itself repeatedly in a recursive process to substitute all straight lines for a right-angled pair.

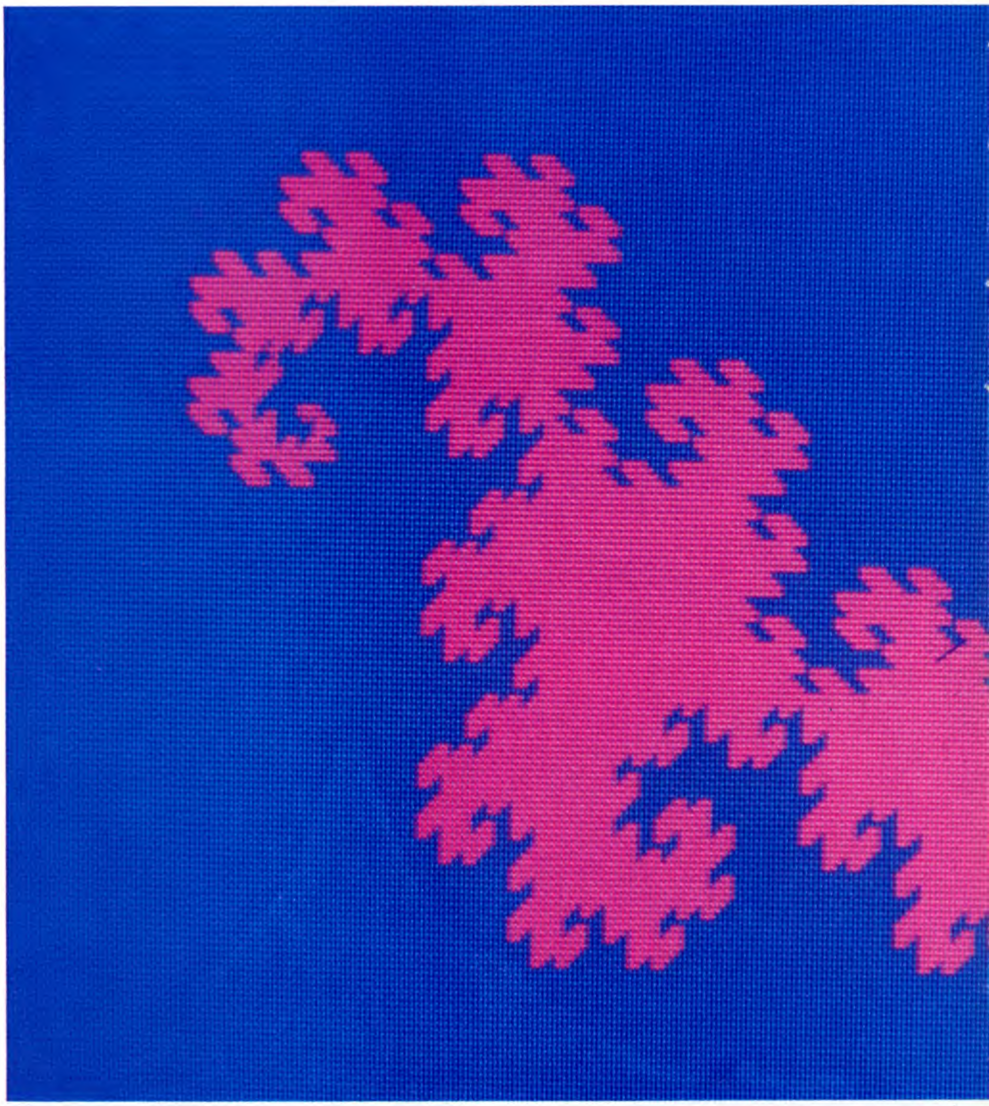
The variable at Line 2Ø is set to the length of the shortest line, so the program recurses until this value is reached. Try changing the value at Line 2Ø to see the effect. Smaller values increase the number of recursive levels, so the time for execution increases. But if you enter large numbers, the program speeds up, yet you can see the details of how the curve develops.

If you watch this growth process carefully, you can see how the process of self-similarity means that any section of the curve looks like a smaller version of the main curve itself. Even so, it is not easy to predict the final shape of the curve.

DRAWING DRAGONS

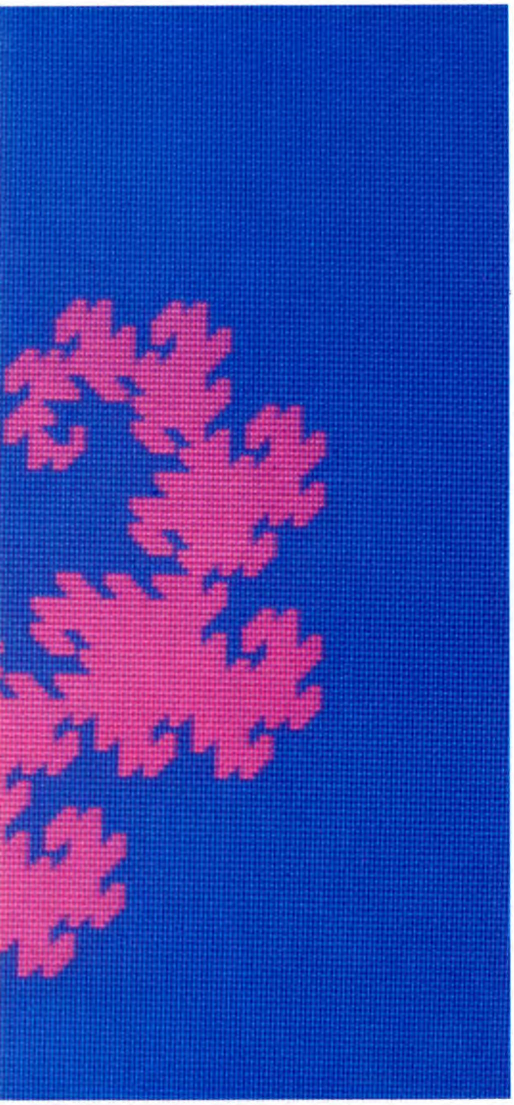
As often happens with fractals, and especially those drawn by recursion, remarkably few lines of programming are required to produce even complicated shapes. Enter and RUN the next program to see a dragon curve, which is another example of the C-curve technique.

In the previous program, each line is replaced by a right-angled pair in the same orientation—always on the same side of the line. In this program, however, the right-angled pair is placed first on one side of the first line, then on the opposite side of the next line, and so on till the shape is completed:



```
1Ø BORDER Ø:PAPER Ø:INK 7:CLS :CLEAR
   3ØØØØ:LET S=32767:DEF FN
   A(X)=(X/8-INT (X/8))*8
2Ø LET MN=1:LET A=Ø
3Ø LET C=ATN (1)/45:DIM S(1Ø):DIM C(1Ø)
4Ø FOR I=1 TO 8:LET S(I)=SIN A
5Ø LET C(I)=COS A:LET A=A+PI/4:NEXT I
6Ø LET L=128:LET X=52:LET Y=8Ø:LET
   T=-1:POKE S,T+1:LET S=S-1
65 DRAW INVERSE 1;OVER 1;X-PEEK
   23677,Y-PEEK 23678
7Ø GOSUB 1ØØØ
8Ø STOP
1ØØØ LET L=L/1.414
1Ø1Ø IF L<MN THEN LET L=L*1.414:LET
   X=X+(L*C(I)):LET
   Y=Y-(L*S(I)):DRAW X-PEEK
   23677,Y-PEEK 23678:RETURN
1Ø2Ø LET I=FN A(I+T):POKE S,T+1:LET
   S=S-1:LET T=1:GOSUB 1ØØØ:LET
   S=S+1:LET T=(PEEK S)-1
1Ø3Ø LET I=FN A(I-2*T):POKE S,T+1:LET
   S=S-1:LET T=-1:GOSUB 1ØØØ
1Ø4Ø LET S=S+1:LET T=(PEEK S)-1:LET
   I=FN A(I+T):LET L=L*1.414:RETURN
```

```
1Ø HIRES Ø,1:MULTI 2,8,9:COLOUR 7,7:
   S=16384
2Ø MN=1
3Ø C=ATN(1)/45
4Ø FOR I=Ø TO 7:S(I)=SIN(A)
5Ø C(I)=COS(A):A=A+π/4:NEXT I
6Ø L=8Ø:X=3Ø:Y=1ØØ:T=-1:POKE S,
   T+1:S=S-1
65 XX=X:YY=Y
7Ø GOSUB 1ØØØ
8Ø GOTO 8Ø
1ØØØ L=L/1.414
1Ø1Ø IF L>=MN THEN 1Ø2Ø
1Ø12 L=L*1.414:X=X+(L*C(I)):
   Y=Y-(L*S(I)):LINE XX,YY,X,Y,
   RND(1)*3+1:XX=X:YY=Y
```



```
1Ø14 RETURN
1Ø2Ø I=(I+T)AND7:POKE S,T+1:S=S-1:
   T=1:GOSUB 1ØØØ:S=S+1:T=PEEK
   (S)-1
1Ø3Ø I=(I-2*T)AND7:POKE S,T+1:
   S=S-1:T=-1:GOSUB 1ØØØ
1Ø4Ø S=S+1:T=PEEK(S)-1:I=(I+T)
   AND7:L=L*1.414:RETURN
```

```
1Ø GRAPHIC 2:COLOR 6,6,3,3:S=4Ø96
2ØMN=1
3Ø C=ATN(1)/45
4Ø FOR I=Ø TO 7:S(I)=SIN(A)*18
5Ø C(I)=COS(A)*18:A=A+π/4:NEXT I
6Ø L=35:X=15Ø:Y=5ØØ:T=-1:POKE S,
   T+1:S=S-1
65 POINT Ø,X,Y
7Ø GOSUB 1ØØØ
8Ø GOTO 8Ø
1ØØØ L=L/1.414:REGION (X AND 5)
1Ø1Ø IF L>=MN THEN 1Ø2Ø
1Ø12 L=L*1.414:X=X+(L*C(I)):
   Y=Y-(L*S(I)):DRAW 1 TO X,Y
1Ø14 RETURN
1Ø2Ø I=(I+T)AND7:POKE S,T+1:S=S-1:
   T=1:GOSUB 1ØØØ:S=S+1:T=PEEK
   (S)-1
1Ø3Ø I=(I-2*T)AND7:POKE S,T+1:
   S=S-1:T=-1:GOSUB 1ØØØ
1Ø4Ø S=S+1:T=PEEK(S)-1:I=(I+T)
   AND7:L=L*1.414:RETURN
```

```
1Ø MODE1: VDU 23; 82Ø2; Ø; Ø; Ø; :M=9
2Ø DIM  S(7)  ,C(7)
3Ø FOR T= ØTO 7: S(T)= SIN (T*PI/4):
   C(T)= COS (T*PI/4): NEXT
4Ø MOVE 18Ø, 6ØØ
5Ø PROCDRAW (792, Ø, -1)
6Ø END
1ØØØ DEF  PROCDRAW (L, I, T)
1Ø1Ø IF L<M□THEN PLOT1,L*C(I), L*S
   (I): ENDPROC
1Ø2Ø L=L/1.414
1Ø3Ø PROCDRAW (L,  (I+T) AND 7,1)
1Ø4Ø PROCDRAW (L, (I-T) AND7, -1)
1Ø5Ø ENDPROC
```

```
1Ø PMODE4:PCLS:SCREEN1,1:CLEAR2ØØ,
   3ØØØØ:S=32767
2Ø MN=1
3Ø C=ATN(1)/45:PI=4*ATN(1)
4Ø FORI=ØTO7:S(I)=SIN(A)
5Ø C(I)=COS(A):A=A+PI/4:NEXT
6Ø L=128:X=52:Y=8Ø:T=-1:POKES,
   T+1:S=S-1
65 LINE-(X,Y),PRESET
7Ø GOSUB1ØØØ
8Ø GOTO8Ø
1ØØØ L=L/1.414
1Ø1Ø IF L<MN THEN L=L*1.414:
   X=X+(L*C(I)):Y=Y-(L*S(I)):
   LINE-(X,Y),PSET:RETURN
1Ø2Ø I=(I+T)AND7:POKES,T+1:S=S-1:
   T=1:GOSUB1ØØØ:S=S+1:T=PEEK
   (S)-1
1Ø3Ø I=(I-2*T)AND7:POKES,T+1:
   S=S-1:T=-1:GOSUB1ØØØ
1Ø4Ø S=S+1:T=PEEK(S)-1:I=(I+T)
   AND7:L=L*1.414:RETURN
```

Apart from the different fractal element that it uses, the main differences between this and the previous program are to do with the speed of execution and the way the element is drawn. Instead of working out the angles each time the element is drawn, this new program calculates the sine and cosine of the angles once, Lines 4Ø and 5Ø, then stores them in two arrays. Fetching values from arrays is much quicker than having to calculate them each time a line is replaced, with the result that the program executes rapidly.

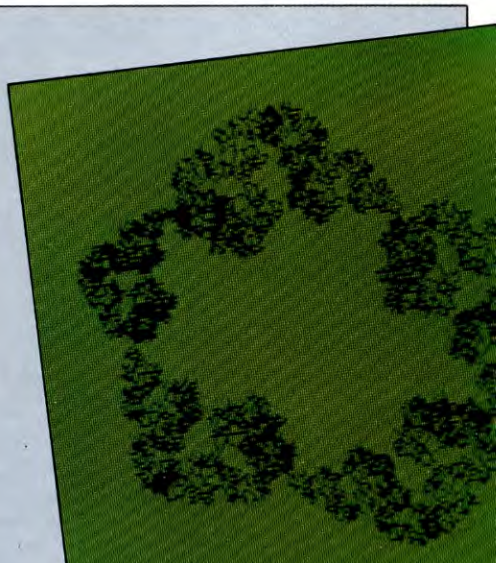
You might argue that the C-curve and dragon curve are little more than mathematical curiosities. As such, they are attractive to look at and fascinating to study—and once you begin to alter the values to which variables in these listings are set, you should be able to produce a range of interesting variations on the basic shapes. The second article on fractals will show you how to introduce uncertainty into drawing mathematical shapes to create more lifelike natural models.

Experimental models

In the second article on fractals, which follows on pages 1434 to 1439, you will see how the principles covered here can be used to extend your graphics repertoire.

The most important difference between the mathematically generated fractals covered here and the shapes that occur in nature is that natural forms possess elements of disorder and irregularity.

The next series of programs shows how you can add random factors to your fractal generation that will mimic these natural variations. And so that you can explore the effects of using different basic shapes and variables, there is also a multi-purpose fractal generator that allows you to pick your own seed values. You'll get a different shape each time you try it, and just one of the many possible is shown in the picture on the right.



CLIFFHANGER: TAKE A RUNNING JUMP

Willie is now fighting fit. He's moving about and jumping up and down. And this episode will see him leaping up the cliff like a vigorous young mountain goat

This is the final part of the series of routines that make Willie move. Now he will be able to dodge out of the way of boulders, jump up the inclines, walk along the flat bits and jump over the potholes and snakes.



```
     org 59472
mfj  cp 129
     jr nz,mfb
     inc a
     ld (57335),a
     ld hl,(57332)
     ld de,22561
     add hl,de
     ld a,(hl)
     cp 43
     jp z,mdy
     cp 44
     jr nz,mfa
     ld hl,(57332)
     dec hl
     ld (57332),hl
mfa  ld hl,(57332)
     ld de, 32
     sbc hl,de
     ld (57332),hl
     ld bc,57Ø72
     ld de,515
     ld a,4Ø
     call 5897Ø
     ret
mfb  cp 132
     jr z,mfd
     inc a
     ld (57335),a
     ld a,(57334)
     cp 1
     jr z,mfc
     ld hl,(57332)
     ld bc,16384
     ld a,45
     ld de,515
     call 5897Ø
     inc hl
     ld (57332),hl
     ld bc,57ØØØ
     ld a,4Ø
     ld de,258
     call 5897Ø
     ld a,1
     ld (57334),a
     ret
mfc  ld hl,(57332)
     ld bc,57Ø16
     ld a,4Ø
     ld de,514
     call 5897Ø
     inc hl
     ld (57332),hl
     ld de,22592
     add hl,de
     ld a,(hl)
     cp 44
     jr nz,mcf
     ld a,Ø
     ld (57335),a
     ld a,3
     ld b,5
     call scr
mcf  ld a,Ø
     ld (57334),a
     ret
mfd  ld a,Ø
     ld (57335),a
     ld hl,(57332)
     dec hl
     ld bc,16384
     ld a,45
     ld de,514
     call 5897Ø
     ld de,33
     add hl,de
     ld (57332),hl
     jp 59153
     org 599ØØ
scn  ret
     org 5933Ø
mdy  *
```

The first time the processor reaches this routine the accumulator will have the number 129 in it. This is the number stored in memory location 57,335 by the first part of the man-moving routine when both the M and the N keys were pressed.

UP, UP AND AWAY

The first thing this routine does is check that 129 is in the accumulator. The only alternatives are 13Ø, 131 or 132 and if these are found **cp 129** and **jr nz,mfb** jumps the processor on to the label **mfb**. But the first time, the accumulator contains 129.

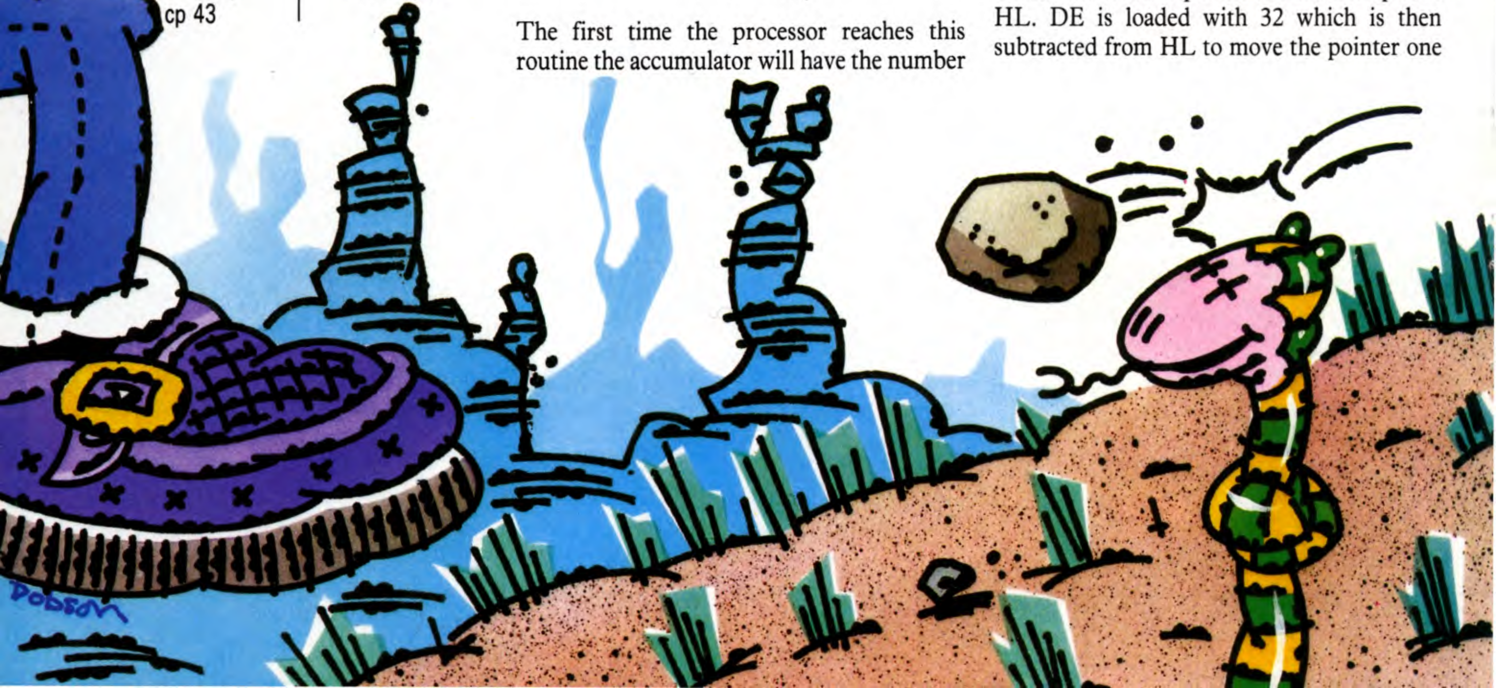
The first thing that it does is to increment the accumulator and load it back into 57,335.

The **ld hl,(57332)** loads Willie's position out of 57,332 into HL. DE is loaded with 22,561 and added to HL to get the address of the attribute of the position in front of Willie's foot. The instruction **ld a,(hl)** then loads the attribute into the accumulator.

This is then compared to 43, the attribute of the snake. If this is found, Willie will have jumped onto the snake's tongue, been bitten and died. So the **jp z,mdy** instruction takes the processor to the **mdy** death routine.

If Willie's not dead, the processor compares the attribute in front of his foot to 44, the attribute for the ground. If ground is not found, the **jr nz,mfa** jumps the processor onto the **mfa** routine. If ground is found, Willie's position is decremented.

At **mfa** Willie's position is loaded up into HL. DE is loaded with 32 which is then subtracted from HL to move the pointer one



- FORWARD JUMPS
- CHECKING FOR SAFE LANDING PLACES
- INCREMENTING SCORE
- RESETTING VARIABLES

character square up the screen. This value is loaded back into 57,332.

BC is loaded with 57,Ø72, the start address of the data for one of the pictures of Willie jumping forward. DE is loaded with 515 to specify that a 2 by 3 block is printed $2 \times 256 + 3 = 515$. The attribute for Willie's colour—4Ø for blue on cyan—is loaded into A and the block print routine is called.

Next time this routine is called the accumulator will have 13Ø in it. So it will skip the first part and jump on to the label **mfb**.

This little routine begins with a different sort of test. The contents of the accumulator are compared to 132 and a forward jump to **mfd** is made if 132 is found. So if the accumulator contains 19Ø or 131, the **mfb** routine is performed.

The first thing this routine does is increment A and load the result back into 57,335.

Next the man-mode flag is loaded into the accumulator from memory location 57,334. This tells the processor which of the two pictures of Willie is required.

The contents of the accumulator are then compared to 1. If 1 is found the processor jumps onto the label **mfc** and prints up the picture of Willie with his legs apart.

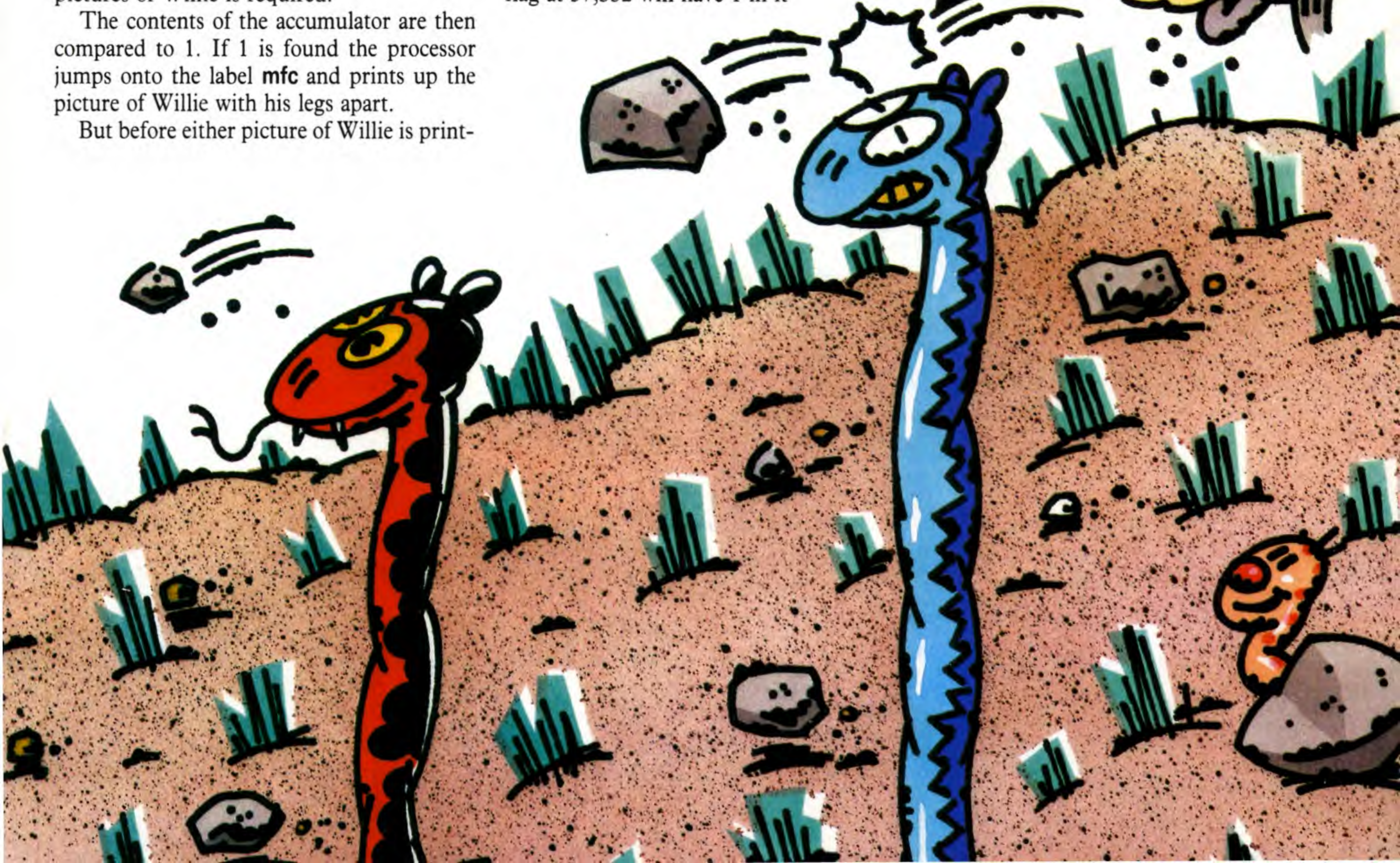
But before either picture of Willie is printed up, any remnants of the last picture have to be obscured. So Willie's position is loaded up into HL from 57,332. BC is loaded with 16,384, the beginning of the screen. A is loaded with 45 to set the colour to cyan on cyan. And DE is loaded with 515 to give a block two by three—$2 \times 256 + 3 = 515$. Then the block print routine at 58,97Ø is called to overprint Willie with sky.

HL is then incremented to move the screen pointer onto the next character square. It is loaded back into 57,332 to update the pointer there too. BC is loaded with 57,ØØØ which is the start address of the data for the picture of Willie with his legs together. A is set to 4Ø—blue on cyan. DE is set to 258—one by two.

After Willie's picture has been printed, A is loaded with 1 which is then stored back in 57,334 so that, next time, the other picture of Willie will be printed.

WILLIE II

Next time the routine is called, the man-mode flag at 57,332 will have 1 in it so the processor will jump directly to the label **mfc**. Here HL will be loaded up with Willie's screen position again. BC will be loaded with 57,Ø16, the start address of the data for the picture of Willie with his legs apart. A is set to 4Ø again, and DE is 515—Willie with his legs apart takes up a 2×2 block. The block print routine is then called to print him up on the screen again.



You'll notice that HL has not been incremented in this part of the routine. This is because Willie is now straddling two character squares. So he has moved forward roughly half a square.

But once Willie has been printed up, the screen pointer in HL and 57,332 is incremented, ready to print Willie further forward.

HIGHER GROUND

Next you need to check whether Willie has managed to clear a slope. So DE is loaded with 22,592 which is added to the screen position in HL. This then points to the attribute of the position under Willie's front foot. The **ld a,(hl)** loads this attribute into the accumulator where it is compared to 44.

If what's in A is not the colour of the ground, Willie has not managed to jump up a section of the slope—he's in mid-air somewhere—and the **jr nz,mcf** instruction sends the processor forward to **mcf**.

But if it is the colour of ground, Willie has hopped up a level, which earns him extra points. Firstly, though, you have to switch the jump off.

The jump is switched off simply by loading Ø into A and storing it back in the jump counter in location 57,335. The number 3 is then loaded into A and 5 is loaded into B. These parameters are going to be fed into another routine called **sci**—or **sc**ore **i**ncrement. In fact, 5Ø is going to be added to the score. The tens column is three columns from the left and five is going to be added to it. Although **sci** is called here, it hasn't been written yet so a **ret** is being added at its start address.

Once the score has been incremented, A is loaded with Ø and stored back in 57,334. This happens whether Willie has landed on higher ground or not. It ensures that the next picture of Willie to be printed up will be the one which has his legs together.

HAPPY LANDINGS

Once both pictures of Willie flying have been printed up on the screen—in whichever order—the man-jump variable in 57,335 will have been incremented up to 132. So next time the man-moving routine is called, the processor jumps all the way to the last short routine which starts at the label **mfc**.

This starts off by setting the man-jump variable back to zero. Ø is loaded into A and stored in 57,335. Then Willie's position is decremented.

BC is loaded with 16,384, the start of the screen. A is set to 45, cyan on cyan. And DE gets 514 for a 2 × 2 block. Calling the block print routine then prints over Willie.

DE is then loaded with 33 which is added to HL. This moves the screen pointer down one line and across one character square to the place where you want Willie to be when he lands. The result is loaded back into 57,332.

```
        ORG 25088
        LDA #230
        STA $07F8
        LDA $C005
        BEQ AA
        LDA #231
        STA $07F8
        JSR $5300
        DEC $C005
        RTS
AA      LDA $C006
        BEQ BB
        LDA #230
        STA $07F8
        JSR $5450
        DEC $C006
        RTS
BB      JSR $5350
        CMP #32
        BNE CC
        JSR $5400
        RTS
CC      JSR $5200
        LDA $0384
        CMP #3
        BEQ DD
        CMP #2
        BNE EE
        JSR $5450
        LDA #4
        STA $C006
        JMP FF
EE      CMP #1
        BNE RET
        JSR $5350
        LDA $FB
        SEC
        SBC #39
        STA $FB
        LDA $FC
        SBC #0
        STA $FC
        LDY #0
        LDA ($FB),Y
        CMP #32
        BNE RET
        JSR MOVERT
        RTS
DD      JSR $5450
        LDA #4
        STA $C005
FF      LDA #2
        STA $0384
        JSR $6850
        JSR $6800
RET     RTS
MOVERT  LDA $D000
        CLC
        ADC #4
        STA $D000
        LDA #0
        ROL A
        STA $0384
        LDA $D010
        EOR $0384
        STA $D010
        INC $C012
        LDA #1
        STA $0384
        JSR $6850
        JSR $6800
        RTS
```

The number 23Ø is loaded up into the accumulator and stored in the pointer for the zero sprite data. Remember that this number is effectively multiplied by 64. Sprite zero will take its data from memory location 23Ø × 64 (= 14,72Ø) onwards. This is where the data for the picture of Willie standing is located.

WILLIE JUMPERS

A is then loaded with the contents of $CØØ5. This is a flag which tells routine whether a jump to the right is required and it is set later in this routine.

If $CØØ5 contains Ø—and a jump to the right is not required—the BEQ instruction following it takes the processor on to label AA. But if it is set—that is, it's not Ø—the processor continues. It loads 231—the data for Willie jumping—into $Ø7F8, the zero sprite pointer. Then the processor jumps to the subroutine at $53ØØ which moves Willie's sprite—sprite zero—up and to the right.

That done, Willie has launched himself into flight and the flag in $CØØ5 can be decremented again.

If the flag in $CØØ5 was not set, the processor branches direct to AA where the flag in $CØØ6 is checked. This is the flag which is set if Willie is going to jump up and down on the spot. If it is not set, the processor moves onto the label BB.

But if it is, the zero sprite pointer is loaded with the effective address of the data for Willie standing again. Then the processor jumps to the subroutine at $545Ø which prints Willie up in the air. When it returns the flag in $CØØ6 is decremented.

TERRA FIRMA

If neither of these flags are set, the processor ends up at BB and looks to see whether there is ground beneath Willie's feet. As the two little routines above turn off their flags, it could be that Willie is in mid flight. If so it is time that he's landed.

First, the processor jumps to the subroutine at $535Ø which checks to see what is underneath Willie's feet. The appropriate byte is returned in the accumulator and compared with 32, ASCII for a space. If what is below Willie's feet is not a space, the BNE branches forward to the label CC.

If there is a space below Willie's feet, the branch is not made and the processor jumps to $54ØØ which moves him down one space.

RUNNING UP THE FLAGS

You may have wondered where the jump flags in the routines above were set. The answer is in the next bit of programming.

If Willie is happily standing on a bit of terra firma—and not an empty space—the processor branches forward to the label CC. There it is sent off to the subroutine at $52ØØ which checks the joystick and keyboard.

That routine returns the action required in memory location $Ø384. The number 3 in that location means that both the Z and the shift key have been pressed, so a jump up and to the right is required. A 2 means that just the shift key has been pressed and Willie is only going to jump up and down. And a 1 in that $Ø384 means that just the Z key has been pressed and Willie is walking.

The contents of location $Ø384 are examined by loading them up into the accumulator. First of all they are compared with 3. If a 3 is found, the BEQ branches to DD.

The contents of $Ø384 are then compared

with 2. If 2 is not found, the BNE instruction takes the processor forward to the label EE.

JACK-IN-THE-BOX

Willie is made to jump up and down like a jack-in-the-box by jumping straight to the subroutine at $5450 which moves the man up four pixels. A is then loaded with 4 and stored in the 'jump-up' flag at $C006. This means that the routine will be called four times.

The processor then jumps to the label FF.

WILLIE WALKING

If the shift key has not been pressed at all and Willie is not being launched into the air—either forwards or just up and down—the processor reaches the label EE. There the contents of memory location $0384, which are still in the accumulator, are compared to 1.

If the contents are not 1, the BNE instruction makes the processor branch forward to the label RET, which marks an RTS. He's standing still and the processor returns.

If he's walking, the first thing to do is check that Willie isn't going to tread on anything nasty. So the processor jumps to the subroutine at $5350 which checks below.

The screen pointer that points to Willie's position is stored in $FB and $FC. The contents of $FB are loaded into the accumulator. The carry flag is set and 39 is subtracted from the accumulator. The result is stored back in $FB. Any 'borrow' is accounted for by loading the contents of $FC into the accumulator, subtracting 0 from it and loading the result back into $FC.

This moves the pointer to the character square in front of Willie.

The contents of this character square are loaded into the accumulator by the indirect addressed instruction LDA ($FB),Y. But first, Y has to be set to 0. The contents of the accumulator are then compared to 32, the ASCII for a space. If a space is not found, Willie has come up against an incline, so he can't go forward without jumping. In that case, the BNE RET sends the processor off to a return, leaving Willie in the same place.

But if there is a space in front of him, the processor continues and jumps to MANRT.

WILLIE THE BOUNDER

If both the Z key and the shift key have been pressed and Willie is supposed to bound along, the processor arrives at the label DD. There, the processor jumps off to the subroutine at $5450 which moves Willie up the screen by half a character square.

When the processor returns from that, 4 is loaded into the accumulator and stored in the 'jump right' flag in memory location 49,157. This means that the routine which moves Willie up and right will be called four times.

The accumulator is then loaded with 2 and stored in memory location $0384. This location is used to pass parameters through to the sound effects routine at $6950. This routine hasn't been added yet. But when it has been, a 2 in memory location $0384 will give Willie's bounding sound effect.

Willie also scores a point for moving. So after his sound effect, the processor is sent to the routine at $6800 which to increment the score.

WILLIE, BY THE RIGHT!

The X coordinate of the zero sprite—the sprite that carries the picture of Willie—is loaded into the accumulator. The carry flag is cleared and 4 is added. This moves the sprite four pixels to the right.

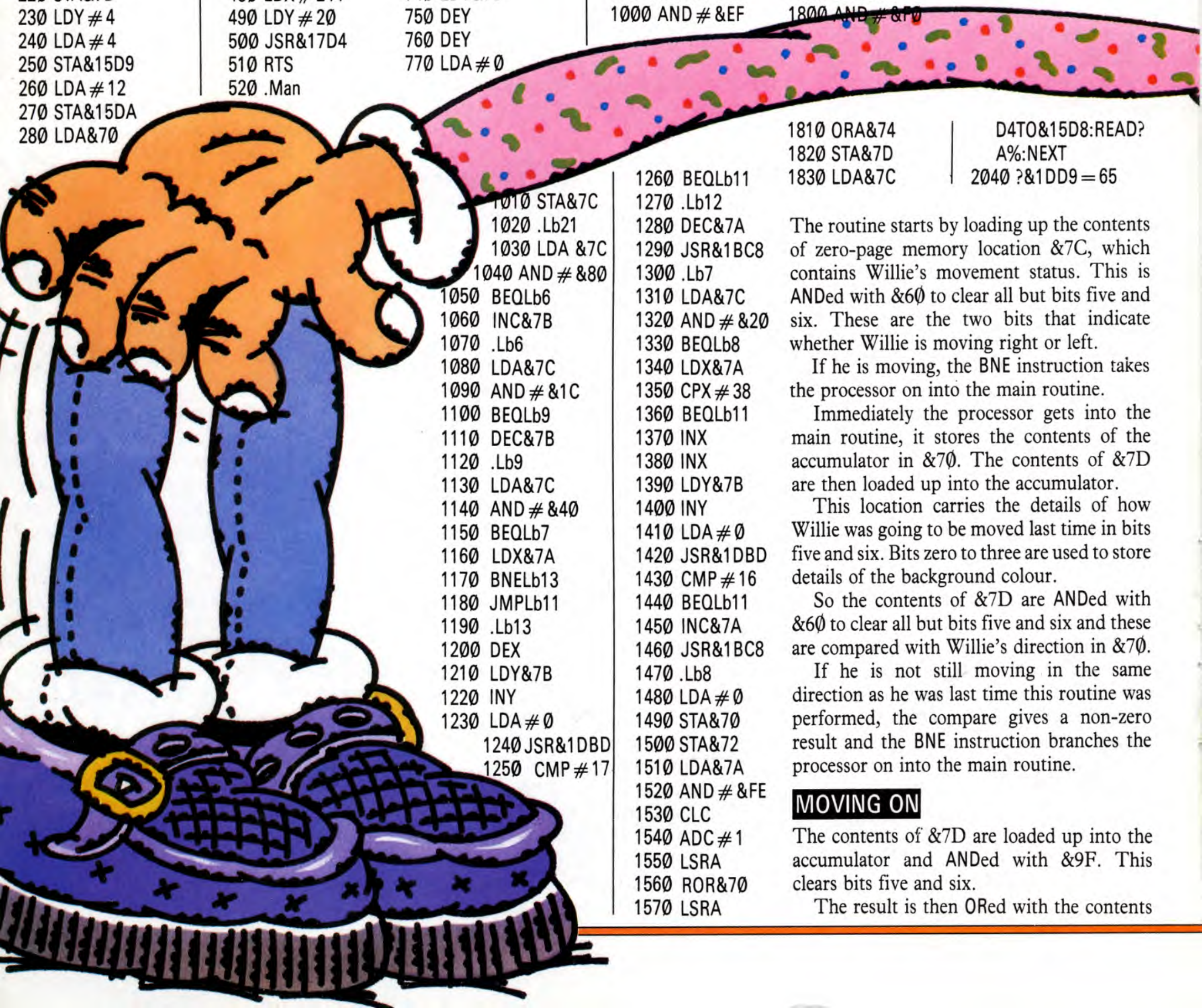
But, of course, you then have to check whether Willie has gone over the 256 barrier and the most significant bit has to be set in $D010. To do this the accumulator is loaded with 0 and any overflow from the addition to the X coordinate in the carry flag is rotated into the zero bit by the ROL instruction. The result is stored temporarily in $0384.

The MSB byte in $D010 is then loaded into the accumulator and Exclusively ORed with the contents of $0384. If the X coordinate has been incremented over the 256 barrier, the EOR puts a 1 in the zero bit. If not, it doesn't. Either way the EOR makes sure that the other bits, which carry the most significant bits of the X coordinates of the other sprites, are left unchanged. The result is stored in $D010.

A is then loaded with 1 which is stored in $0384. The sound effects routine at $6850 and the score routine at $6800 are then called. These make Willie's walking sound and increment his score for moving.



```
30 FORI=0TO3STEP3
40 P%=&1F82
50 [OPTI
60 .Defman
70 LDA&7C
80 AND#&60
90 BNELb2
100 RTS
110 .Lb2
120 STA&70
130 LDA&7D
140 AND#&60
150 CMP&70
160 BNELb1
170 RTS
180 .Lb1
190 LDA&7D
200 AND#&9F
210 ORA&70
220 STA&7D
230 LDY#4
240 LDA#4
250 STA&15D9
260 LDA#12
270 STA&15DA
280 LDA&70
290 AND#&40
300 BNELb3
310 LDY#25
320 LDA#8
330 STA&15D9
340 LDA#4
350 STA&15DA
360 .Lb3
370 LDX#228
380 .Lb4
390 STX&72
400 STY&73
410 JSR&17D4
420 LDX&72
430 LDY&73
440 INX
450 INY
460 CPX#237
470 BNELb4
480 LDX#244
490 LDY#20
500 JSR&17D4
510 RTS
520 .Man
530 JSR&1E99
540 LDA&7C
550 AND#&4
560 BEQLb5
570 DEC&7B
580 JSR&1E99
590 RTS
600 .Lb5
610 JSR&1EEE
620 LDA&7C
630 AND#&4
640 BEQLb14
650 JSR&1515
660 .Lb14
670 LDA&7C
680 AND#&50
690 CMP#&50
700 BNELb20
710 LDX&7A
720 DEX
730 DEX
740 LDY&7B
750 DEY
760 DEY
770 LDA#0
780 JSR&1DBD
790 CMP #17
800 BNE Lb20
810 LDA&7C
820 AND#&EF
830 STA&7C
840 .Lb20
850 LDA&7C
860 AND #&30
870 CMP #&30
880 BNELb21
890 LDX&7A
900 INX
910 INX
920 LDY&7B
930 DEY
940 DEY
950 LDA#0
960 JSR&1DBD
970 CMP#16
980 BNELb21
990 LDA&7C
1000 AND#&EF
1010 STA&7C
1020 .Lb21
1030 LDA &7C
1040 AND#&80
1050 BEQLb6
1060 INC&7B
1070 .Lb6
1080 LDA&7C
1090 AND#&1C
1100 BEQLb9
1110 DEC&7B
1120 .Lb9
1130 LDA&7C
1140 AND#&40
1150 BEQLb7
1160 LDX&7A
1170 BNELb13
1180 JMPLb11
1190 .Lb13
1200 DEX
1210 LDY&7B
1220 INY
1230 LDA#0
1240 JSR&1DBD
1250 CMP#17
1260 BEQLb11
1270 .Lb12
1280 DEC&7A
1290 JSR&1BC8
1300 .Lb7
1310 LDA&7C
1320 AND#&20
1330 BEQLb8
1340 LDX&7A
1350 CPX#38
1360 BEQLb11
1370 INX
1380 INX
1390 LDY&7B
1400 INY
1410 LDA#0
1420 JSR&1DBD
1430 CMP#16
1440 BEQLb11
1450 INC&7A
1460 JSR&1BC8
1470 .Lb8
1480 LDA#0
1490 STA&70
1500 STA&72
1510 LDA&7A
1520 AND#&FE
1530 CLC
1540 ADC#1
1550 LSRA
1560 ROR&70
1570 LSRA
1580 ROR&70
1590 LSRA
1600 ROR&70
1610 STA&71
1620 LDA&7B
1630 AND#&FE
1640 SEC
1650 SBC#1
1660 LSRA
1670 ROR&72
1680 LSRA
1690 ROR&72
1700 LSRA
1710 ROR&72
1720 LSRA
1730 ROR&72
1740 STA&73
1750 LDY#&0
1760 LDX#&70
1770 LDA#9
1780 JSR&FFF1
1790 LDA&7D
1800 AND#&F0
1810 ORA&74
1820 STA&7D
1830 LDA&7C
1840 AND#&F8
1850 BEQLb10
1860 LDA&7C
1870 EOR#1
1880 STA&7C
1890 .Lb10
1900 JSRDefman
1910 JSR&1E99
1920 RTS
1930 .Lb11
1940 LDA&7C
1950 AND#&9F
1960 STA&7C
1970 JMPLb8
1980 ]NEXT
1990 ?&1582=12
2000 ?&1583=0
2010 ?&1585=&41
2020 DATA 255,35,1,
  2, 4, 4, 12, 0
2030 FORA%=&15
  D4TO&15D8:READ?
  A%:NEXT
2040 ?&1DD9=65
```



The routine starts by loading up the contents of zero-page memory location &7C, which contains Willie's movement status. This is ANDed with &60 to clear all but bits five and six. These are the two bits that indicate whether Willie is moving right or left.

If he is moving, the BNE instruction takes the processor on into the main routine.

Immediately the processor gets into the main routine, it stores the contents of the accumulator in &70. The contents of &7D are then loaded up into the accumulator.

This location carries the details of how Willie was going to be moved last time in bits five and six. Bits zero to three are used to store details of the background colour.

So the contents of &7D are ANDed with &60 to clear all but bits five and six and these are compared with Willie's direction in &70.

If he is not still moving in the same direction as he was last time this routine was performed, the compare gives a non-zero result and the BNE instruction branches the processor on into the main routine.

MOVING ON

The contents of &7D are loaded up into the accumulator and ANDed with &9F. This clears bits five and six.

The result is then ORed with the contents

of &7Ø, this updates them to the direction Willie is moving in now. And the result is then stored back in &7D so that it can be referred to next time this routine is called.

The next thing to do is redefine the characters that make up Willie so that he is facing the direction he is walking in. First of all, you assume he is going left and new data is POKEd into &15D9 and &15DA which are used to define UDG character 244.

The contents of &7Ø are then loaded up into the accumulator again and ANDed with &4Ø. This isolates bit six, which is the one that tells Willie to go left. If Willie is going left, this bit will be set and the BNE will branch over the next little routine.

If it is not set, Willie is moving right, the branch is not made and the processor continues with the next little routine which POKEs different data into &15D9 and &15DA. This data makes Willie face right.

Next the UDG characters from 228 to 236 are redefined by the loop in Lines 38Ø to 47Ø. The X register carries the UDG number and the Y value depends on whether Willie is moving left or moving right.

The routine at &17D4, given on page 1Ø37, is used to redefine the characters. X and Y are incremented each time the processor goes round the loop, but they have to be stored in &72 and &73 when &17D4 is called. That routine uses the X and Y registers for its own purposes. And the CPX and BNE instruction in Lines 46Ø and 47Ø let the processor drop out when 236 has been redefined.

Next UDG 244 itself has to be redefined. X is loaded with the UDG number, 244, and Y is loaded with 2Ø. The processor then jumps to the subroutine at &17D4.

UPS AND DOWNS

The next part of the routine concerns moving Willie, now that he is facing in the right direction. The first thing it does is to jump to &1E99 which erases the last Willie.

Then the contents of &7C are loaded up and ANDed with &4. This clears all the bits except bit two. If this is not set, Willie is alive and BEQ branches over the next routine.

If it is set, Willie is dead and the branch is not made. In that case, Willie's Y coordinate in &7B is decremented, which moves him one line down the screen, and the subroutine at &1E99, printing Willie in his new position.

If Willie was still alive the processor was branched forward to Line 6ØØ. It then jumped to &1EEE to check for key presses.

The next routine—in Lines 68Ø to 1Ø2Ø—checks whether Willie has landed on the edge of a slope in a half character position.

If he is not dead, the BNE instruction branches the processor forward again. The contents of &7C are then ANDed with &8Ø, which isolates bit seven. This is the bit that tells Willie whether to jump. If he is jumping, the instruction in Line 1Ø6Ø increments Willie's Y coordinate in &7B.

Next the contents of &7C are ANDed with &1C, to check whether Willie is falling. If he is, the instruction in Line 75Ø decrements Willie's Y coordinate in &7B.

WILLIE'S GAUCHE

Then the contents of &7C are ANDed with &4Ø, to check whether Willie is moving to the left. If he is not moving to the left, the BEQ branches the processor forward. If he is, it loads Willie's X coordinate into X.

If Willie's X coordinate is zero, he has already reached the edge of the screen and you don't want him to go any further. So the BNE instruction branches over the next instruction if the X coordinate is not zero. If it is zero the processor proceeds to the JMP instruction in Line 118Ø and jumps forward almost to the end of the routine at Line 193Ø.

If Willie was not at the edge of the screen, the X pointer is decremented by the DEX instruction in Line 12ØØ. Y is loaded with the contents of &7B and incremented. A is loaded with Ø. And the processor jumps to the subroutine at &1DBD. This looks at the character square in front of Willie's feet and returns the value of it in A.

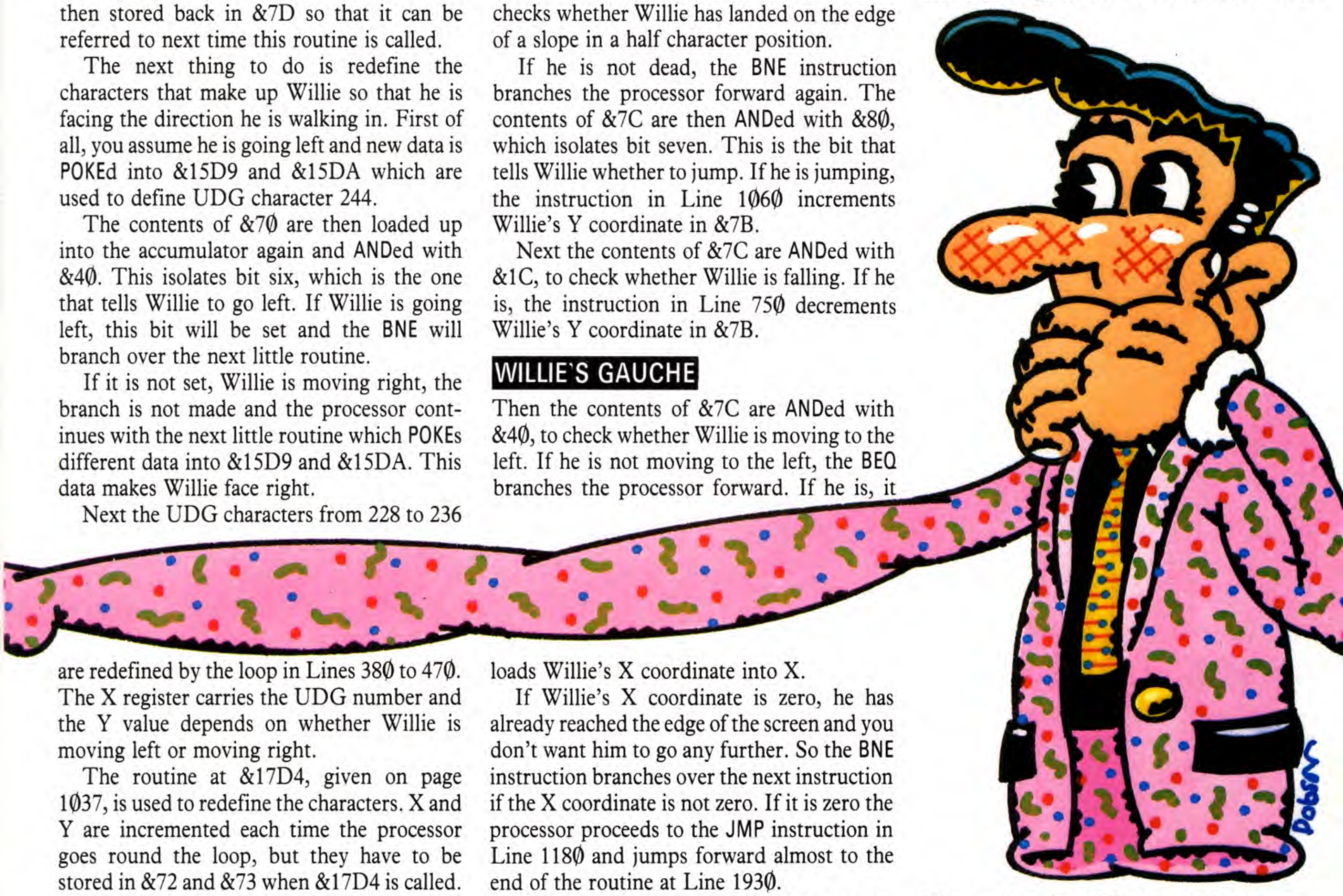
Comparing the result with 17, in Line 125Ø, asks whether the square in front of Willie is a slope. If it is, Willie can move no further without jumping. So the BEQ instruction sends the processor to the end of the routine.

If the square in front is not land, Willie's X coordinate in &7A is decremented, moving him to the left. Then the processor jumps to &1BC8 which makes a walking sound.

WILLIE'S RIGHTS

The contents of &7C are loaded up yet again in Line 131Ø and ANDed with &2Ø. This isolates bit five. If it is not set, Willie is not moving to the right and the processor jumps forward over the next routine. But if it is set, Willie is moving right and it proceeds.

Willie's X coordinate is loaded from &7A into the X register and it is compared to 38. If



it has reached 38, Willie has reached the edge of the screen, so the BEQ takes the processor to the end of the routine.

Otherwise, the X pointer is incremented twice. Y is loaded with Willie's Y coordinate from &7B and incremented. A is loaded with Ø. And the processor jumps to the subroutine at &1DBD. This is the one that looks at the character square in front of Willie's foot and returns its value in the accumulator.

Comparing this with 16 checks whether it is the other slope graphic. If it is, Willie can proceed no further without jumping and the BEQ sends the processor to the end.

Otherwise, the X coordinate in &7A is incremented to move Willie one character square to the right. Then the processor jumps to &1BC8 again.

FILLING IN THE BACKGROUND

When examining the contents of a screen location—when doing a collision check, for example—you must know the background colour of the character square in question. The routine in Lines 147Ø to 182Ø looks at a

pixel in the middle of the block, gets its colour value and stores it in bits zero to three of &7D.

The routine in Lines 143Ø to 161Ø takes the X coordinate and multiplies it by 32 and stores the result in &7Ø and &71. And the routine in Lines 162Ø to 174Ø takes the Y coordinate and multiplies it by 16 and stores in the results in &72 and &73. This changes character-squares into pixel coordinates.

To find out what colour the pixel at that point is, the OSWORD routine at &FFF1 is called with a 9 in the accumulator. The contents of X and Y have to be the low and high bytes of the base address of the coordinate table respectively. Here the data table is on the zero page, so Y is loaded with Ø. And X is loaded with &7Ø which is the start of the pixel coordinates you have just worked out.

When &FFF1 is called, the result is returned in the next location after the end of the data table—in this case, &74. So when the OSWORD routine has been performed, the contents of &7D are loaded up into the accumulator and ANDed with &FØ. This clears bits zero to three. The result is then ORed with the contents of &74—the pixel colour—and stored back in &7D.

LEGGING IT

The contents of &7C are loaded up once again into the accumulator and ANDed with &F8. This clears bits one, two and three and checks whether Willie is moving at all.

If he is not, the BEQ instruction branches the processor forward over the next little routine. But if he is moving, he must be moving his legs. So the processor proceeds and loads up the accumulator with the contents of &7C one more time. This is then Exclusively ORed with 1, to flip bit zero, and the result is stored back in &7C.

The processor then jumps to the subroutine, labelled Def man, given earlier in this part of Cliffhanger. It makes Willie face the right way. Then it jumps to the subroutine at &1E99 which prints Willie.

STANDING STILL

So far throughout this routine the processor has been branched forward to the label Lb11 in Line 193Ø. This has occurred each time Willie has run into the slope or the edge of the screen and can proceed no further.

The contents of &7C are loaded up for the last time in this part of Cliffhanger and ANDed with &9F. This clears bits five and six which are the ones that control Willie's left and right movement. The result is stored back in &7C.

The processor then jumps back to the label Lb8 in Line 147Ø, does the background colour check and prints up Willie.

```
        ORG 2Ø321
MFJ     CMPA #129
        BNE MFB
        INC 18261
        LDX 18249
        LEAX 29Ø,X
        LDA ,X
        CMPA #$57
        LBEQ MDY
        JSR MFZ
        LDX 18249
        LEAX -255,X
        STX 18249
        LDU #17814
        JSR CHARPR
        LEAX 254,X
        LDU #17846
        JSR CHARPR
        RTS
MFB     CMPA #13Ø
        BNE MFC
        INC 18261
        JSR MFZ
        LDX 18249
        LEAX -255,X
        STX 18249
        LDU #1787Ø
        JSR CHARPR
        LEAX 254,X
        JSR CHARPR
        LEAX 254,X
        JSR CHARPR
        LDX 18249
        LEAX 864,X
        LDA ,X+
        CMPA #$FF
        BEQ MFF
        LDA ,X
        CMPA #$FF
        BEQ MFF
        RTS
MFF     LDA #4
        LDB #5
        JSR SCI
        RTS
MFC     CMPA #131
        BNE MFD
        INC 18261
        JSR MFZ
        LEAX 254,X
        LDU #1536
        JSR CHARPR
        LDX 18249
        LEAX 257,X
        STX 18249
        LDU #17814
        JSR CHARPR
        LEAX 254,X
        LDU #17846
        JSR CHARPR
        RTS
MFD     CMPA #132
        BNE MFE
        JSR MFZ
        LDX 18249
        PSHS X
        LEAX 512,X
        LDY ,X
        PULS X
        CMPY #$5555
        BNE MFE
        LEAX 1,X
        STX 18249
MFE     CLR 18261
        CLR 18251
        LDX 18249
        LDU #17774
        JSR CHARPR
        LEAX 254,X
        JSR CHARPR
        RTS
MFZ     LDX 18249
        LDU #1536
        JSR CHARPR
        LEAX 254,X
        JSR CHARPR
        RTS
CHARPR EQU 194Ø2
SCI     EQU 2Ø751
MDY     EQU 2Ø126
```

The first time the processor reaches this routine the accumulator will have the number 129 in it. This is the number stored in memory location 18,261 by the first part of the man-moving routine when both the M and the N keys were pressed.

WILLIE AWAY

The first thing this routine does is check that 129 is in the accumulator. At this point if it is not, 13Ø, 131 or 132 must be and the CMPA #129 and BNE MFB branch the processor on to the label MFB. But the first time the processor hits this routine, A contains 129.

The first thing that it does is to increment the accumulator and load it back into 18,261, so that the processor will go onto the next part of this routine next time it is called.

The LDX 18249 loads Willie's position out of 18,249 into X. 29Ø is added to this to get a byte from in front of Willie's feet. LDA ,X loads this into the accumulator and CMPA #$57 compares it with $57, the colour graphic for a snake's tongue. If this is directly in front of Willie when he jumps, he will be bitten and die and the LBEQ MDY will take the processor off to the death routine.

If Willie is still alive, the processor is sent off to perform the MFZ subroutine, which prints sky on the two character squares under Willie. You don't want his feet left there.

Willie's position is loaded up from 18,249 into X again. And it is decremented by 255 to move it up and across the screen one character square. The new position of the screen pointer is then stored back in 18,249.

U is then loaded up with 17,814. This is the start address of the data for the picture of Willie with his legs apart. U, remember, is used as the data pointer for the CHARPR routine, making the data the user stack.

CHARPR is called. X is incremented by 254 to move it down the screen to the start position of Willie's bottom half. U is loaded with 17,846, the beginning of the data for Willie's bottom half. And CHARPR is called again. This prints Willie above the cliff.

I'M WILLIE, FLY ME

Next time the man-moving routine is called the accumulator will have 13Ø in it and the MFB routine will be performed. On subsequent calls, when the accumulator will be carrying a higher value, the CMPA #13Ø and BNE MFC instructions will send the processor on to the next part of the program.

Again, the first thing the routine does is increment the contents of 18,261. Then a jump to MFZ blanks Willie's bottom half.

Next Willie's position is loaded up into X again and decremented by 255 once more. This moves the screen pointer up and across one more character square. The U is loaded with 17,87Ø and CHARPR is called three times with X being incremented by 254 between each call. This prints up a third picture of Willie that spans three character squares.

Willie, in fact, has not got any taller. He has simply spilled over into an extra character square because he's shifted up half a square.

BEATING THE BOULDER

Willie gets extra points if he successfully clears a boulder while he is jumping. So you have to check if a boulder is under him.

Willie's position is loaded from 18,249 into the X register. This is then incremented by 864 to look underneath Willie. A is then loaded with the byte of the screen pointed to by X and X is incremented.

The screen byte in A is compared to $FF, the colour graphic of the boulder. If it is found, the processor branches forward to the label MFF. If not, the next screen byte is loaded up and compared. And if the boulder is found there, again the processor branches forward to MFF. If not, it returns.

The MFF routine updates the score. A is loaded with 4. This specifies that it is the fourth digit from the left—and tens—that is going to be updated. B is loaded with 5. This is the number of time that it is going to be updated. Then the processor jumps to a routine called SCI, a scoring subroutine which will add 5Ø to the score. You have not added this routine yet. So, to prevent the program crashing, put an RTS in the start location.

LEVEL FLIGHT

The next time the man-moving routine is called the accumulator will carry 131. This is checked for. And if it is not found the Willie is in a later stage of his jump and the processor is branched forward to the label MFD.

Again, the first thing the routine does is increment the man jump variable in 18,261 so that next time the routine is called the processor will proceed direct to the next part.

Next, the processor jumps to the MFZ subroutine which blanks out the top and middle part of Willie. His feet are overprinted with sky by the next short routine. This extra bit of screen clearing has to be done because Willie was three square's tall last time.

The position in X which was set during the MFZ routine is incremented by 254, moving the pointer down to Willie's feet. The data pointer in U is loaded with 1536, the address of blank sky in the top left-hand corner of the screen. CHARPR is called to do the overprinting.

Willie's old screen position is then loaded up into X again from 18,249 and incremented by 257. This moves it down the screen across one character.

The data pointer in U is then loaded with the start address of the picture of Willie with his legs apart at 17,814 again. The CHARPR subroutine then prints up his top half. The screen pointer in X is incremented to move it down the screen one line and U is loaded with the address of the data for Willie's bottom half. Then CHARPR is called to print it up.

LEVEL GROUND

A 132 in the accumulator carries the processor forward to the routine where Willie lands on the ground again. And if a 132 is not found in A, the processor branches forward to the label MFE. This should never happen. If the contents of A are any less than 132, the processor should have stopped skipping forward at an earlier routine. And as the man-jump variable in 18,261—which A is loaded from before the routine selection starts—is not incremented again. Still, this is a good failsafe device. If any rogue number has got into 18,261, the processor branches direct the clear instruction that sets 18,261 back to zero.

If the number in A is 132, the processor continues with this routine. Firstly, it jumps off to the MFZ subroutine to clear the last picture of Willie. Then Willie's position is loaded from 18,249 back into X.

The screen pointer in X is pushed onto the stack to save it, then it is incremented by 512 so that it points to the character squares under Willie's feet. The two bytes there are loaded into the Y register and the original screen pointer is pulled back off the stack into X.

The contents of Y are compared to $5555, the screen graphic of a blank yellow square in other words, a hole. If one is not found the processor jumps forward to the label MFE.

But if there is a hole there Willie needs a bit of help. Because of the way the Dragon and Tandy's screens are laid out, it is more than likely that Willie will land in a hole which would make the game impossible to play. So X is incremented to push Willie over the hole and the result is loaded back into his position pointer in 18,249.

You'll note that there is no check to see whether Willie has landed on a higher level. This is because, when he jumps up and forward, he always lands on a higher level.

CLEARING UP

The jump is now complete. But you have to clear the jump counter and the location that carries Willie's picture number. These are set back to zero so that the processor will go to the walk routine in part one of these man-moving articles and start Willie off with the right picture.

You finish off by printing a picture of Willie standing still. X is loaded with his position from 18,248 and U is loaded with 17,774 and CHARPR is called twice.

The last routine is MFZ which is called to blank Willie out. X is loaded with Willie's position from 18,249 and U is loaded with 1536, which is blank sky. To see program work, try the following:

```
10 POKE30000,57:POKE20847,57:POKE
   20751,57
20 EXEC19426
30 FORL=1TO8:POKE18261,129:FORJ=1TO
   5:EXEC19902
40 FORK=1TO100:NEXTK,J,L
50 GOTO50
```



LISP—THE LANGUAGE OF LISTS

Although slow to start, LISP has gathered speed and 'LISP machines' may well be the prototypes of the home computers on sale during the next ten to twenty years

LISP was originally developed by John McCarthy at the Massachusetts Institute of Technology in the early 1960s, a time when the most sophisticated computer languages were primitive versions of FORTRAN—a language like BASIC used by scientists and engineers. The name is derived from LISt Processing—which gives a clue to the way in which the language works. Today, twenty-five years later, LISP programs are still based on the same fundamental ideas while FORTRAN has had to take on new facilities as people have gradually come to understand writing programs better.

Even if you have never heard of LISP before, the programs you write will probably have been influenced by it because many of its concepts have been applied to other languages. Despite its influence on other languages, however, the ideas of LISP are still radically different to those of languages like BASIC or Pascal, and LISP is a considerably more powerful language than both of these.

THE IMPORTANCE OF SPEED

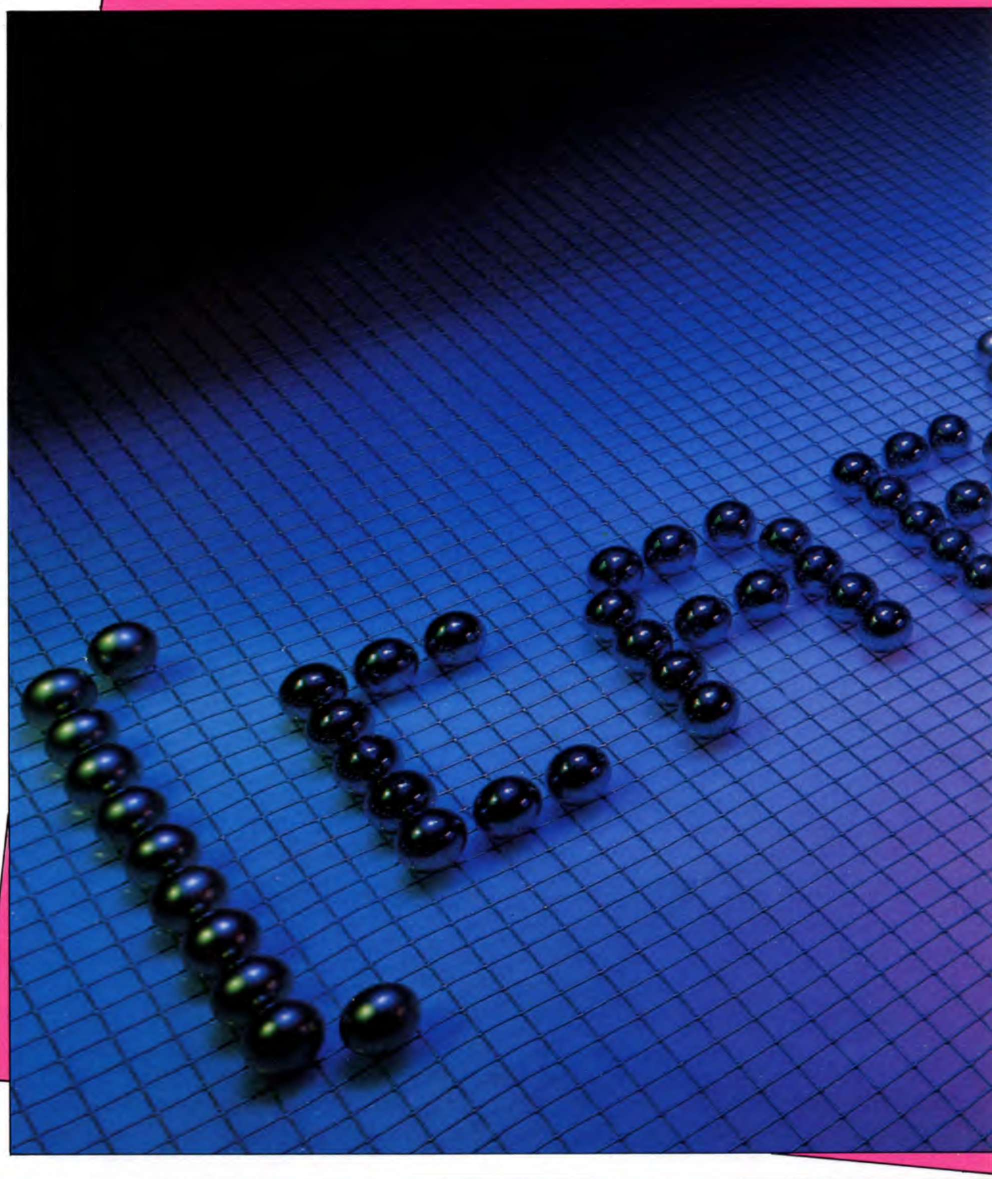
It is well known that it is easier to write a program in BASIC to perform a given task than it is to write the same program in machine code. BASIC is a higher level language and the features in it act as a sort of intellectual lever, allowing the programmer to tackle more difficult problems.

In the future, much more complex tasks are going to be done by computers than at present, and this means that programmers are going to have to use even more powerful languages like LISP in order to write the programs, but powerful languages tend to be slower.

It is certainly true that early versions of LISP were slow, and this held the popularity of the language back. However, present day versions of LISP that run on large computers are as fast as any other high level language. Furthermore, the next product of the revolution in integrated circuit technology will be a vast increase in the processing speed and power that can be fitted onto a silicon chip. This will mean that it will no longer be vital to squeeze the last bit of speed out of a system by using machine code.

ARTIFICIAL INTELLIGENCE

LISP is the main language used in Artificial Intelligence. The aim here is to create computer programs that exhibit some kind of human behaviour or intelligence. An example is the chess-playing computer. Another is one of the most famous examples of the use of LISP. This was a program called ELIZA which played the role of a psychoanalyist talking to its 'patients' by means of a keyboard and screen display. The program simulated the conversation that might take place, asking and answering questions. Some people were actually fooled into thinking that the computer really understood what they



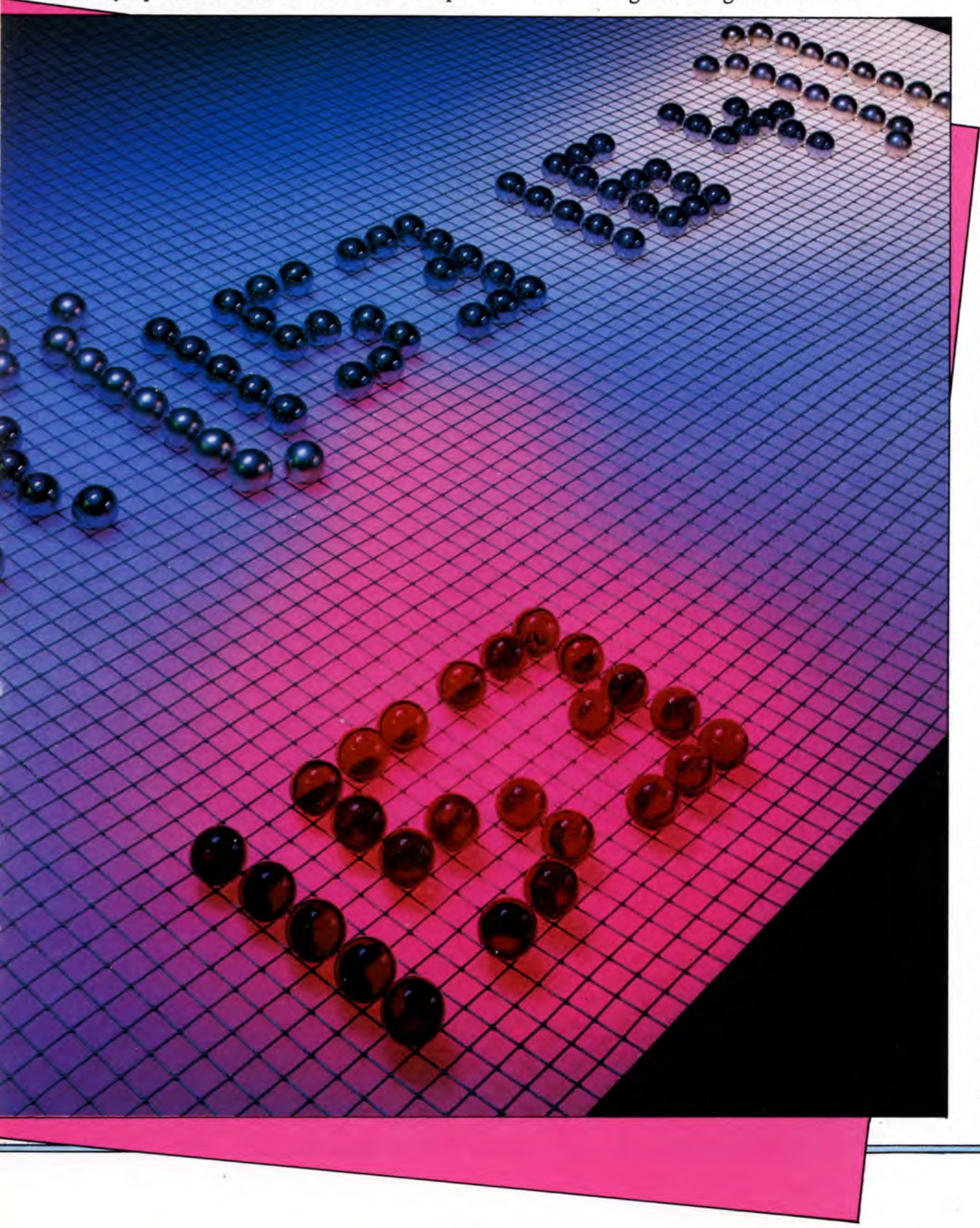
- THE IMPORTANCE OF SPEED
- ARTIFICIAL INTELLIGENCE
- OTHER APPLICATIONS
- LISP AND THE COMPUTER
- THE FUNDAMENTALS
- LOADED WITH LISP
- ATOM VALUES
- NUMBERS IN LISP
- TRUE OR FALSE
- KEEPING TRACK

told it. In fact, the program simply looked at its input and replied by suitably modifying it. The analysis necessary to do this is much easier in LISP than in other languages.

A further area of artificial intelligence is that of Expert Systems. The idea here is to program into a computer all the knowledge of a human expert in some field such as motor car maintenance. Untrained users can then find out what is wrong with a car, by typing in the symptoms of the faults. The computer will reply with its diagnosis, asking any further questions that might be necessary in order to locate the problem precisely.

OTHER APPLICATIONS

Other, more traditional complicated programs like compilers and word processors are much simpler when written in LISP and the language has also been used to write some of the Computer Aided Design systems used in the design of integrated circuits.



LISP has also played a distinguished part in computer programs that do algebra. It is obvious that computers are very good at arithmetic. However, with LISP it is possible to make them do algebra as well. This means that computers can make the kind of algebraic calculations that would occupy an army of mathematicians and cover miles of paper.

Some of the calculations in theoretical physics contain algebra so complicated that it can only be done by such LISP-based systems. On a less complex level, a typical application of LISP that you could run on your home computer would be a small LISP program that could do differentiation—very useful if you have to plough through dozens of calculus examples for O and A level homework.

LISP AND THE COMPUTER

It would be wrong to suggest that LISP is the only language suitable for solving difficult problems or likely to be important in the future. Several modern languages have appeared which use its concepts widely, including LOGO and PROLOG. And often the compilers and interpreters for these languages were originally written in LISP. To some extent therefore, LISP is the assembly language of the new generation of computer languages. Nowadays, there are, in fact, special processors whose assembly language is based on LISP in much the same way as the Z8Ø, 6502 or 6809 processor in your home computer understands the appropriate machine codes.

These LISP processors often form the heart of special high power personal computers or engineering work stations whose main language is LISP. Such 'LISP machines' may well be the prototypes of the home computers on sale in the high streets in five or ten years time.

These articles on LISP, explain the fundamentals of the language and you will be able to follow what is going on even if you don't have LISP yet. Just like BASIC, there are differences between the LISP language on the different computers, but fortunately, the variations in LISP are usually quite small—normally only consisting of different spellings

for function names. A bad point about LISP is that there are an awful lot of brackets. If you are a person who has difficulty getting the brackets in BASIC expressions to balance, LISP may not be the language for you.

THE FUNDAMENTALS

One of the things that separates LISP from BASIC, is that LISP can manipulate symbols as well as numbers. This is something like the way BASIC can slice and manipulate strings, but the symbol manipulation capabilities of LISP are much more powerful.

The basic objects that LISP handles are called *atoms* and *lists*. An atom is really like a variable in BASIC—that is it starts with a letter and may then continue with any length (in practice there *is* a limit) of letters and numbers. So all the following could be atoms: FRED, atom, NO1, NO2. Everyone is familiar with many kinds of list—for example, a shopping list or a team list of the players in a game. Your shopping list could be: tea, sugar, milk. A list in LISP is very similar—including a number of atoms and perhaps other lists. To tell LISP that something is a list, it is surrounded by a pair of brackets. So your shopping list in LISP would look like:

```
(tea sugar milk)
```

A LISP list can contain both atoms and lists so another example of a list is:

```
((cold tea) (hot milk) sugar).
```

This list contains two lists and an atom. Brackets are used very much like the way double quotes are used in BASIC to show where strings start and end. The difference is that strings cannot contain further strings.

LOADED WITH LISP

If you want to try LISP out for yourself, there is no need to trade in your basic home computer for a £30,000 LISP machine. In fact, most home computers can be converted to LISP. The only reason a computer understands BASIC is that there is large program in read only memory (ROM) which is a BASIC interpreter. It is perfectly possible to replace this machine code with another program that interprets or understands LISP programs. In practice, on some computers adding LISP actually will consist of plugging in a chip or cartridge which contains a ROM with the necessary code in it. More often the cheaper method of holding the machine code LISP interpreter in the computer's main memory is adopted. Here, you load LISP in off a tape or disk like any other program, and when the code is run your home computer becomes a LISP system.

When this happens, what would it look like? Instead of the usual BASIC prompt, you would get the message Evaluate:. This means that LISP is waiting for you to type in a valid LISP expression which it will then evaluate in printing its value on the screen. This is all there is to LISP, you type in *expressions* and the computer replies with their values. In LISP an expression means a symbolic expression consisting of either a list or an atom. Often this will be referred to as an *s-expression*. There is no command like RUN which actually sets off a program. As you will see later, such a command is not needed.

With the Evaluate: prompt waiting, you must decide what to type. The easiest thing, is an atom FRED. LISP replies with UNDEFINED. The dialogue looks like this:

```
Evaluate: FRED
Value is: UNDEFINED
```

This means that LISP has never seen the atom before and cannot find a value for it. Atoms, like variables in BASIC can have values. The situation above is like PRINTing a variable in BASIC before giving it a value. When LISP sees the atom, it tries to evaluate it, or find its value. This is a very important feature of LISP—everything is always evaluated unless there is a specific intention not to do so.

There is a method of stopping evaluation in LISP—the single quote mark '. Suppose you typed 'FRED. Now, LISP would reply with FRED. The single quote mark stops any further evaluation of the s-expression following it. The value of the expression 'FRED is just the atom following the quote. The single quote mark can also be used with lists. Thus:

```
'(tea sugar milk)
```

will evaluate to:

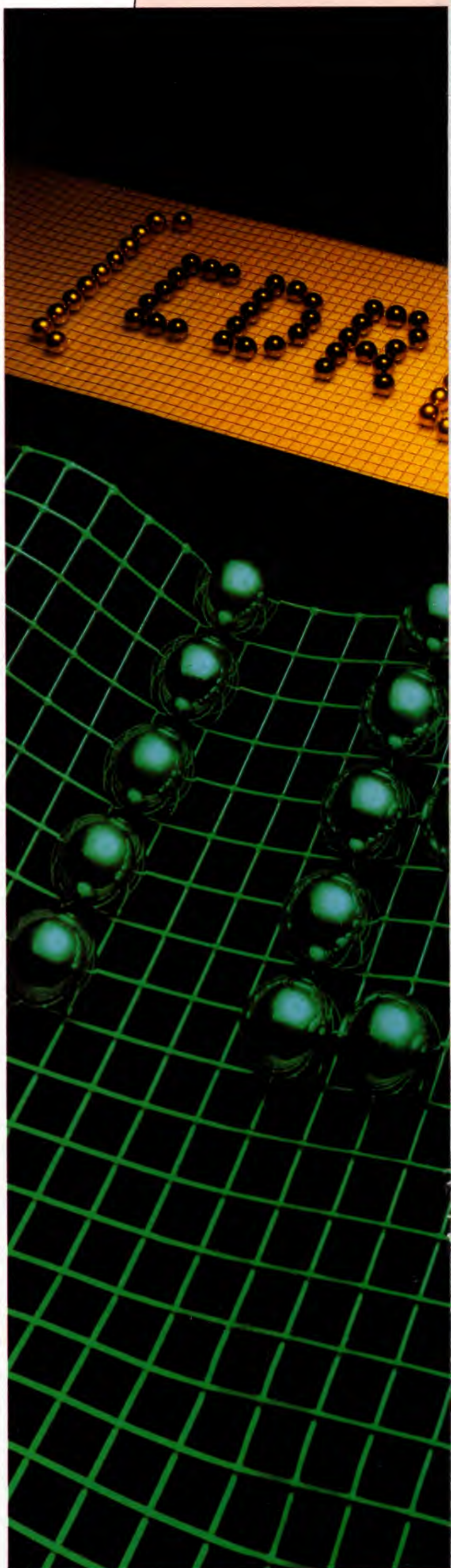
```
(tea sugar milk).
```

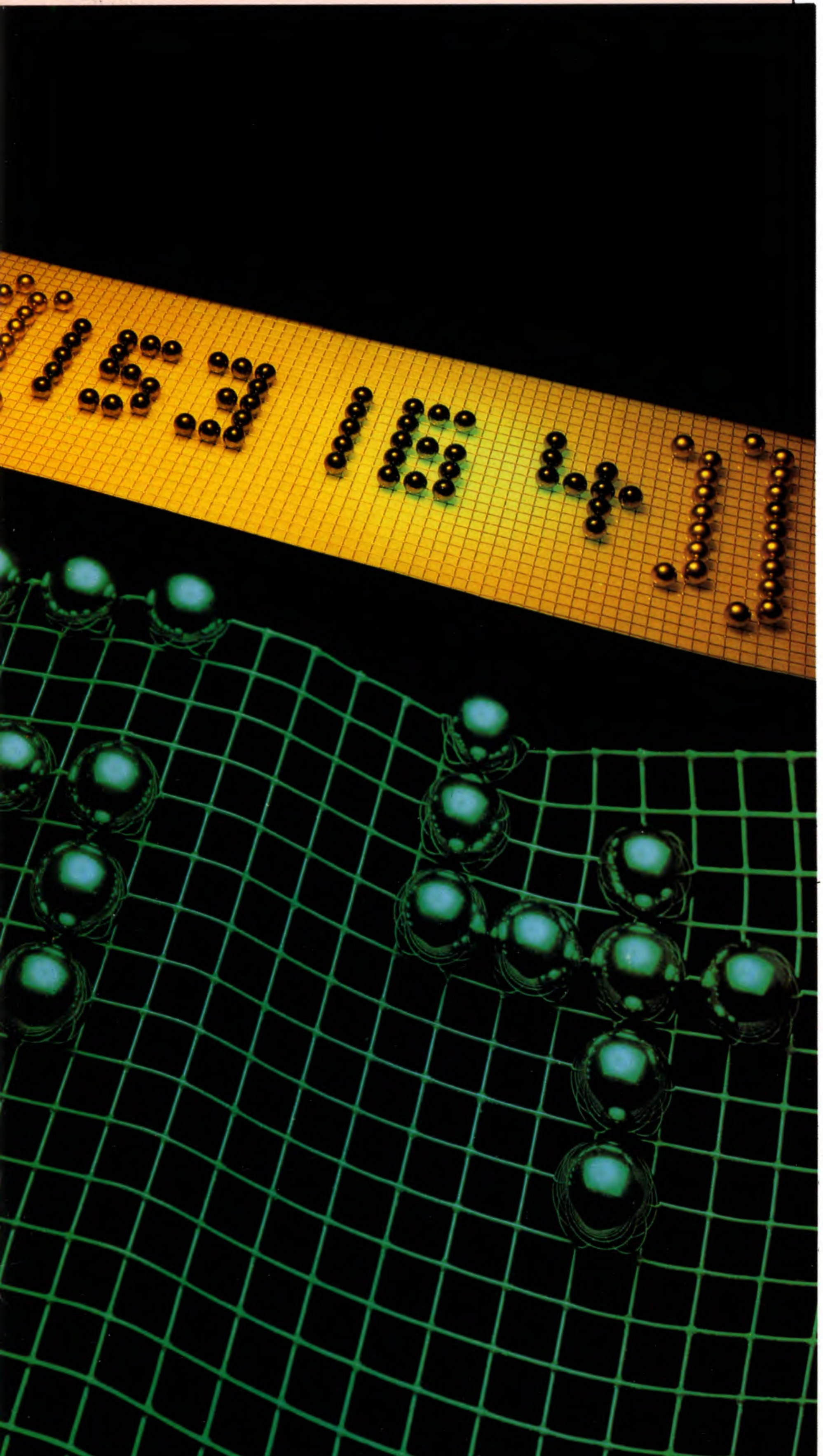
A much more interesting use than the evaluation of atoms is the evaluation of lists. An example of this is again provided by the single quote—'FRED is actually an abbreviation for (QUOTE FRED). Some systems will in fact only accept the second form of the expression. Here QUOTE is a LISP function and the atom FRED is its argument. Note that QUOTE does not find the value of its argument.

Compare this function with functions in BASIC, where you might write:

```
RND(20).
```

BASIC finds that the value of this is a random number. The same thing in LISP would look like:





(RND 2Ø)

Notice that the brackets have been moved. Similarly, if there was a QUOTE function in BASIC, it would be written:

QUOTE(FRED).

This movement of brackets can be confusing but is simple when you understand it.

ATOM VALUES

Atoms, like variables, can have a value. However, the question arises; how does an atom get a value? For instance how do you give FRED the value 1? This is done with the function SET as follows:

(SET 'FRED 1).

As usual, LISP evaluates everything. The first argument of SET evaluates to FRED and the second to 1—the function then sets FRED to 1. This shows two things. First, all numbers in LISP are atoms which evaluate to their numeric value. Second, just like in BASIC, functions can have more than one argument. Now if you typed FRED, LISP would reply with 1. The interaction with LISP could look like:

```
Evaluate: (SET 'FRED 1)
Value is: 1
Evaluate: FRED
Value is: 1
```

Like all other functions, SET must return a value—usually the value of its second argument, in this case 1. There is another version of SET, SETQ which does not evaluate its first argument. So you could type:

(SETQ FRED 1)

with no quote in front of FRED. To hammer home the point about evaluation, suppose you next typed:

(SETQ GEORGE FRED)

LISP would evaluate FRED and find its value was 1 and then set GEORGE to 1. GEORGE is not evaluated and need not be protected by using a '.

The SET function is rather like the way some versions of BASIC have a command LET. In BASIC, variables often have to be of a declared type. For instance, in BBC BASIC, all integers end with a % and in most versions string variables end with a $. If you wrote:

A% = "STRING"

then you would get an error message. In LISP there is no need to specify variables like this—an atom can have as its value a number, another atom or a list. For example:

(SETQ SHOPS '(tea milk sugar))

will assign your shopping list to the atom SHOPS.

By now, you may have noticed another important difference between LISP and BASIC. First consider the list:

(tea milk sugar)

—the shopping list. However, if tea was a function (like SET) then this list would be a LISP expression and could be evaluated. Later, you will see that it is easy to define atoms like tea to be functions. The point is that in LISP, data and programs look exactly the same. This is a powerful feature, it means that one LISP program can manipulate another LISP program as easily as any other data.

In LISP everything is done by functions and finding the value of expressions. But sometimes this is not very convenient. For example, you are not normally interested in the actual value returned by the SET function—it is much more important that it *gives* an atom a value. So some functions are only used for their side effects and not for their value.

CUTTING IT UP

Obviously, to do anything with LISP, you need to be able to manipulate lists—similar to string handling in BASIC, remember the three fundamental functions which do this are called CAR, CDR (pronounced 'cooder' or 'cudder') and CONS. The natural reaction to this piece of information is 'So what?'. But suppose instead that you were told that these functions were called FIRST, REST and ADD to FRONT. These names give you much more information about what they actually do. In fact, CAR and CDR stand for 'contents of address register' and 'contents of decrement register' and come from the names of two of the registers in the original IBM computer on which LISP was used. CONS stands for construct. Most people learn to use these terms, but they could easily be changed by, for example, (SETQ FIRST CAR).

CAR, as suggested by the alternative name FIRST, returns the first element of a list. For instance:

(CAR '(tea milk sugar))

has the value tea. Note that CAR evaluates its arguments so a ' has to be used to prevent evaluation of the list. As another way of seeing how this works, type:

(SETQ SHOPS '(tea milk sugar))

and then:

(CAR SHOPS).

LISP would again give the value of this as tea because it finds the value of SHOPS is the list:

(tea milk sugar).

CDR returns the list that contains all the elements of the list given to it as an argument *except* the first one. So the value of:

(CDR '(tea milk sugar))

is the list:

(milk sugar).

CONS, as its name suggests, constructs lists. It does this in a quite specific way—it has two arguments, an atom or a list and a list. CONS adds its first argument to the beginning of the list which is the second argument. For instance:

(CONS 'biscuits '(tea milk sugar))

evaluates to:

(biscuits tea milk sugar).

For lists this works as follows:

(CONS '(left right) '(hand foot))

has the value:

((left right) hand foot).

The CAR and CDR of this then are:

(left right)

and:

(hand foot).

Note that this time the CAR is a list and not an atom.

Suppose you take the CDR of a list with only one element, for example:

(CDR '(FRED))

This looks as if it might be an empty list (). In fact, LISP has a special name for this—the atom NIL. Taking the CAR or CDR of an atom or an empty list produces an error message as you might expect. There is a generalization of CAR and CDR. Imagine trying to find the atom left in the list:

((left right) hand foot).

You could find this as the value of the expression:

(CAR (CAR '((left right) hand foot)))

LISP provides a shorter way with the function CAAR, and the answer could now be given by:

(CAAR '((left right) hand foot)).

Similarly, CADR means take the CDR and then the CAR. LISP systems provide all the other permutations of CAR and CDR in pairs and often in triples, quadruples and so on.

If you had two atoms FRED and SMITH and wanted to make them into a list, you might be tempted to type:

(CONS 'FRED 'SMITH).

Although this is a valid expression, it is not a list. Its value is the dotted pair:

(FRED . SMITH),

but for the moment this is an unnecessary complication. One way of forming the desired list would be to type:

(CONS 'FRED (CONS 'SMITH NIL)).

The second argument of the first CONS evaluates to the list (SMITH) to which the atom FRED is CONSed onto the beginning. However, LISP provides the rather briefer method:

(LIST 'FRED 'SMITH).

This is actually an example of another LISP feature, functions which can have an arbitrary number of arguments. You could for instance type:

(LIST 'ONE 'TWO 'THREE)

and get the answer:

(ONE TWO THREE).

This is very different from BASIC, where functions always have the same number of arguments.

NUMBERS IN LISP

Although one of LISP's special features is its ability to manipulate symbols, it can also handle numbers. The LISP implementations on home computers usually only use integers. However, some of the big LISP systems can handle floating point numbers and calculate with them as fast as any special purpose numerical language like FORTRAN. A feature often provided on such systems is arbitrary precision arithmetic where any number of significant figures can be kept—perfect for calculating the value of pi to a few million decimal places.

Compared to BASIC, LISP deals with arithmetic in a rather strange way. It is, however, similar to LOGO, covered earlier in the course. The equivalents of the usual BASIC operators +, −, *, / and MOD (which gives the remainder after an integer division) are the functions PLUS, DIFFERENCE, TIMES, QUOTIENT and REMAINDER. The big difference is that LISP uses *prefix* notation in contrast to BASIC

which uses *infix* notation. Prefix notation is sometimes called Polish notation after the Polish logician Lukasiewicz who proved that this (and reverse-Polish or *postfix* notation) are consistent ways of doing arithmetic.

In BASIC, if you want to add 2 and 2 you write 2+2, while in LISP this becomes (PLUS 2 2). In prefix notation the arithmetical operators come first in expressions. This allows LISP to make the arithmetical operators into functions just like any others. So you could change their names if you wanted. Some of the LISP arithmetic functions like PLUS can take any number of arguments. To give an example:

```
(PLUS 2 2 2 (TIMES 1 2))
```

evaluates to:

2+2+2+1*2=8.

In LISP, the function MINUS must be used to change the sign of a number; for example the value of (MINUS 2) is −2. This contrasts with BASIC where the − symbol is used both for changing the sign and for finding the difference in two numbers.

TRUE OR FALSE

The next thing to look at, is how LISP handles logical expressions and how to branch depending on the value of a logical expression. In BASIC, logical values are treated just like integer arithmetic, typically Ø stands for false and −1 for true (other choices are possible). A logical expression is something like:

A>B.

If A is greater than B, then in BASIC this has the value true otherwise, it has the value false. In LISP, true and false are represented by the two special atoms T and NIL (the same NIL described above as the empty list).

True and false values are given by special functions often called *predicates*. An example is the function ATOM which returns the value T if its argument is an atom and NIL otherwise. Typing:

```
(ATOM 'FRED)
```

will provoke the reply T from LISP whilst:

```
(ATOM '(milk biscuits))
```

evaluates to NIL. Other LISP predicates are GREATERP to test which of two atoms has the greater numerical value and NULL which tests a list to see if it is empty. The predicate EQ tests for the equality of any two atoms:

```
(EQ 'FRED 'FEED)
```

is T and:

```
(EQ 'FRED 'HARRY)
```

is NIL. Actually EQ tests if its two arguments have the same value so:

```
(SETQ F '(jam cakes))
```

then:

```
(SETQ G F)
```

followed by:

```
(EQ G F)
```

will have the value T since, the values of both G and F are the same list. In addition to these there are many other predicates in LISP.

As well as the arithmetic operators, LISP has the usual complement of logical operators. These also use prefix notation. For instance, (OR T T) has the value T, (OR T NIL) has the value T, (AND T NIL) has the value NIL and so on.

In BASIC, the first thing to do once you can evaluate logical expressions, is to branch depending on their value using the IF statement. In LISP, the equivalent of this is provided by the COND function. COND has an arbitrary number of arguments (called clauses) each of which is a list containing a number of s-expressions (lists or atoms). When the COND function is evaluated, each clause is looked at in turn and the first s-expression evaluated. If this has the value T then all the subsequent s-expressions in that clause are evaluated. The value of the COND function is then given as that of the last s-expression in the clause. Otherwise, evaluation proceeds to the next clause. If none of the clauses has a first s-expression that is T then the COND function has the value NIL. Usually, things are arranged so that any none NIL value for a first s-expression will give the same behaviour as a value of T.

A typical COND looks as follows:

```
(COND ((EQ FRED 1)(PRINT 'ONE))
      ((EQ FRED 2)(PRINT 'TWO))
      ((EQ FRED 3)(PRINT 'THREE))
```

In this demonstration, another new function, PRINT has been introduced; here this prints out its argument after evaluating it. Although LISP has a comprehensive set of such printing functions they tend to vary from one system to another.

Now if FRED has been SET to 2 and the above COND is given to LISP, it will print the word TWO on the screen. In fact, the value of the PRINT function is usually the value of its argument, so LISP will return the value of the COND as TWO. It should be clear why this happens. First LISP compares the atom FRED with 1 and finds that they are not equal; this means that the first s-expression of the first clause is NIL. Therefore, LISP moves on to the second clause of the COND. This time the first s-expression has the value T and therefore the second s-expression (PRINT 'TWO) is evaluated; printing TWO on the screen and returning the value of the s-expression (PRINT 'TWO) i.e. TWO.

All this may seem rather complex and it is certainly necessary to keep track of the brackets in such an expression. However, there is a very similar statement in BASIC which makes the operation of COND clear. Look at the following fragment of BASIC:

```
IF FRED=1 THEN PRINT"ONE"
ELSE IF FRED=2 THEN PRINT"TWO"
ELSE IF FRED=3 THEN PRINT"THREE"
```

Such a chain of IF THEN ELSE IF's is precisely equivalent to the COND function. Often such chains in BASIC end with a single ELSE. This means that if none of the IF's are satisfied then the last ELSE will catch all other cases. To give this behaviour in LISP, the COND function is terminated with the clause:

```
(T (LISP code for other cases)).
```

If COND finds that none of its other arguments are satisfied, it comes to this clause and evaluates T which has the value T. So the s-expressions following T are always evaluated.

KEEPING TRACK

By this stage it may be hard to remember what is going on. LISP makes this easier by providing the *object list*. The list contains all the atoms which LISP knows have a value. The function OBLIST or something similar has as its value the object list. Thus to see all the objects known to LISP you type:

```
(OBLIST).
```

Note that although this function has no parameters, it must still be surrounded by brackets. In the object list, you will find all the names of the familiar functions like CAR and CDR. Also if you have done something like:

```
(SETQ FRED 1)
```

you will find the atom FRED added to this list.

The key feature that allows programs to be written in LISP is that as in LOGO it is possible for users to define their own functions. These then take their place in the object list and as far as LISP is concerned, such functions are just as important as the built-in ones. So you can extend LISP to include any features you might want, and in the second part of this series, the mysteries of how to use and define LISP functions will be revealed.

FINISHING THE SYMPHONY

Suppressed Schuberts and budding Beethovens! Now you can pitch in, finish the program and make music. There are also detailed notes on how you can use each of the facilities

This is the final part of the music composer program. Once you've added it to the other two parts you can start to create your own musical masterpieces.

The programs are designed to make the best of the sound on each of the computers, and because the computers vary so much the programs, too, are different. They are all explained in detail below, and in some cases—such as the Commodore, on which you can tailor the sound to your precise requirements—the instructions may seem quite complex. The best thing to do is to sit down with the computer and follow through the instructions trying out each of the commands in turn. You'll soon get the hang of them.

The Spectrum's menu offers seven choices. Try option 1 first to turn the computer into a musical keyboard. The musical notes are arranged in the same way as in the simple music article on page 669. Lower C is Q, middle C is I and top C is the B key. The computer asks you if you want to Extend the last tune or Start a new one, so as you haven't programmed a tune yet, press S in this case. You also have to choose which note length you want for the notes by pressing SYMBOL SHIFT with a number from 1 to 5, to select semi-quaver, quaver, crotchet, minim or semi-breve respectively—the higher the number, the longer the note. You can select the length of each note before you play it or you can play the entire tune on one setting and edit in the correct length later, using option 4.

The computer sounds the notes as you play them and it also stores them in memory so you can replay the whole tune using option 3. You can select the correct tempo for the replay by typing in a number from 1 to 15—the larger the number, the faster the tempo.

Option 2 gives you an alternative way of entering the notes by typing in a simple code. The twelve possible notes—from C through C♯, D and so on up to the next B—are numbered from Ø to 11, and a rest is coded as −4. The octave of each note is selected by a number from 1 to 7, and the number which gives the length of each note corresponds to their real length. So a semi-quaver is 1, a quaver is 2, a crotchet is 4, a minim is 8 and a semi-breve is 16. A rest can be given a length in this range in the same way as a note. The code is very easy to use. For example, note C♯, octave 2, crotchet is entered as 12Ø4, and note B, octave 4, semi-breve is 11416. For rests, −4Ø4 is a crotchet-length rest. The only point to remember is to enter the duration as *two* digits, so use Ø2 for a quaver, not 2.

When you've entered a few notes, press RETURN to see the main menu then choose option 3 to replay it.

At this stage you'll probably find that you want to alter a few notes or perhaps add or delete some. So this time choose option 4 from the menu. You'll be offered 5 choices. You'll need to press D first to find out which notes you want to change—jot down the numbers of the notes, then press E to edit them. To change a note, simply enter its number followed by RETURN then type in the new note using the code described above.

To insert a note, press I, then enter the number of the note *before* the one you want to insert, then enter the new note.

To delete a note, press X, then enter its number. The program shifts all the other notes accordingly to take account of any changes.

When you're happy with a tune, you can save it by choosing option 6 from the menu, and then load it in again with option 7.

The Commodore program lets you take full advantage of the sophisticated sound available from the SID chip—but only if you want to. You can use it to compose a simple tune if that's all that you want, or you can specify exactly the values you want for pulse width, filter frequency, ring modulation and a whole host of other things. It's up to you how you use the program. There are ten choices available from the main menu.

For the moment, it's best to ignore options 1 to 4—sensible parameters have already been set up for you. So choose option Ø, then press P to play without storing the notes (while you're practising) or press P and SHIFT to store the tune. Try storing a few notes, then choose option Ø again but this time press R to replay them.

Option 5 lets you enter the notes by typing in a simple code. Remember that the note number *has* to be entered as two characters—as Ø1, not 1.

Rests can also be entered into the tune by typing −1 followed by the voice, for example −11. A note can be lengthened by following it with a sustain, which is code −2 then the length of the note and the voice. The numbers for the lengths are 1 for quaver, 2 for crotchet, 3 for minim and 4 for semi-breve. So adding −231 turns the previous note into a minim. If you don't enter a sustain the notes are taken as semi-quavers.

When you've entered all the notes, press RETURN to get back to the main menu and try replaying the tune again. You can enter notes in all three voices and these will be replayed simultaneously.

At some stage you're sure to want to edit your tune, so choose option 6 and you'll see another menu. Choose D first to display the notes, and write down the numbers of the notes you want to change. Then press E to edit the note. You have to enter its number then the code for the new note or Ø if you want to delete it entirely.

To insert a note, press I then enter the number of the note *before* the one to be inserted and type in the code for the new note. The program automatically takes account of any changes you make.

When you're happy with the tune you can save it by choosing option 8 from the main menu and load it back again with option 9.

SYNTHESIZED SOUNDS

The other options are more complicated and allow you to change the sound of each voice to mimic different instruments or create other synthesized sound effects. Choose option 1 to display the list of values you can change.

The values on the right are automatically set for voice 1 for you. If you alter any, you can initialize them again using option 7 from the main menu—so don't be afraid to experiment. To alter the values, use the cursor keys

to move the arrow up or down and press RETURN when it is next to the item you want to change. All you do then is to type in the new value. Line up the arrow with RETURN and press RETURN to get back to the menu.

Here's what each of the options do: The waveforms available are Triangular, Sawtooth, Pulsed or Noise. Altering this will give a different quality to the sound and if you select N for noise the notes will sound like a percussion instrument.

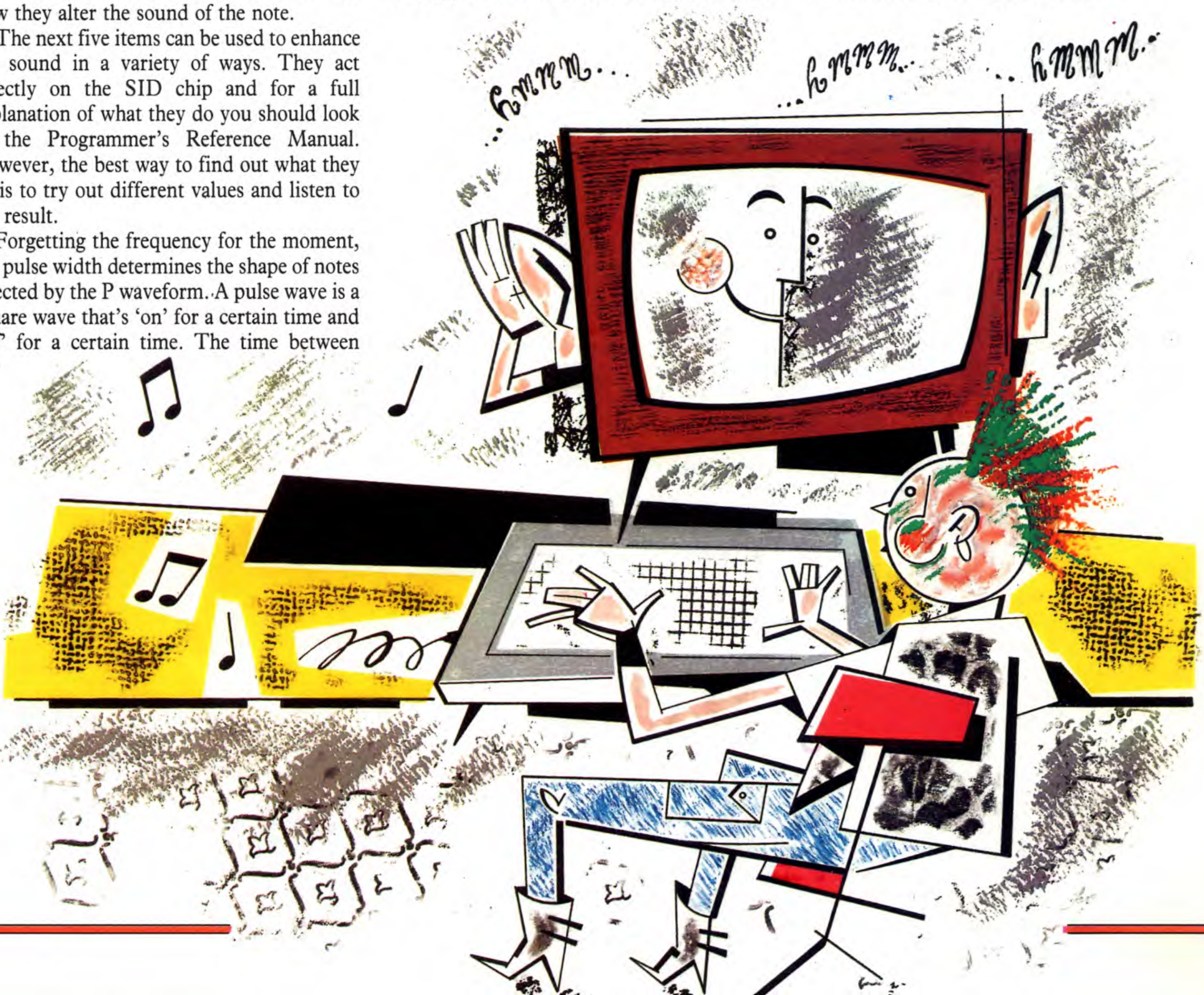
The next four parameters are the A D S R values used to define the shape or envelope of the note. The article on pages 1138 to 1144 explains in detail what these values do and how they alter the sound of the note.

The next five items can be used to enhance the sound in a variety of ways. They act directly on the SID chip and for a full explanation of what they do you should look in the Programmer's Reference Manual. However, the best way to find out what they do is to try out different values and listen to the result.

Forgetting the frequency for the moment, the pulse width determines the shape of notes selected by the P waveform. A pulse wave is a square wave that's 'on' for a certain time and 'off' for a certain time. The time between successive 'ons' is determined by the frequency of the note being played. But the ratio of on time to off time for each cycle is set by the pulse width. A low value, say 500, means it is on for 500 units and off for 3595 units, a ratio of about 1 to 7. A mid value of 2048 gives a perfect square wave which is on and off an equal amount. Try entering different values for pulse width in steps of 500 and listen to the effect on your tune. (Remember to select a P waveform.)

The Synchronization and ring modulation can be turned ON, but you'll only hear an effect by setting the *frequency* in *Voice 3*. This may appear to be a rather odd way of doing it but it is the way the SID chip is designed. (For the other voices, the synchronization and ring modulation in voice 2 is affected by the frequency in voice 1, and that for voice 3 is affected by the frequency in voice 2!) It is difficult to describe the sound produced, so experiment by turning on the synch or ring modulation (or both) in voice 1, then alter the frequency in voice 3 in steps of 10000. Make a note of the values that give the best effect.

Finally, turning the filter ON affects all three voices and filters out the harmonics in the notes. This is most noticeable on the P waveform. Try it with different pulse widths as this will make a difference too.



OVERALL SOUND CHANGES

As well as altering the sound of the individual voices, you can also alter the general sound parameters by choosing option 4. This gives you another menu. Again, the best way to find out what all the options do is to experiment. The filter frequency determines the cut-off point for the filter band passes in the other options.

The low pass lets through all frequencies below the cut off, giving a rather muffled sound. The high pass lets through only the high frequencies above the cut off—making the music sound as though you are listening to it over a telephone. Try setting any of the filters to ON, then alter the filter frequency in steps of about 500 and listen to the result. Also try using different waveforms. With an N waveform, a low pass sounds like a bass drum, a band pass sounds like a side drum and a high pass sounds more like a snare drum or cymbals.

The resonance also alters the quality of the note. The best way to hear it is to choose an N waveform, filter band pass ON, filter frequency of about 1000 and then enter different values for resonance. Also try setting the resonance to 15, and alter the filter frequency.

The voice 3 connection is something provided by the SID chip but has little noticeable effect. Turning it OFF cuts off any low noise from voice 3 that might affect voice 1 when you're using synchronization or ring modulation. It clarifies the sound slightly.

The Vic program is designed to make use of the Super Expander cartridge. This means the program can be very short as all the sound commands needed are already available. The menu displays the range of values you can input for volume, voice, octave, tempo, notes, rests, sharps and flats. Sensible values for the first four have already been set, but to alter them simply type in the letter shown followed by a number in the allowed range. For example, typing V1T1 then RETURN sets a low volume and slow tempo.

The program lets you play a tune by pressing the keys A to G. You can choose whether the notes sound as you press them, and you can choose whether the computer stores the notes or whether you just want to practise. The rule is, if the notes are printed on the screen then they are being stored, if they're not printed they are not being stored.

The program stores the tune as strings of notes. Each string can hold up to four lines of notes as they appear on the screen, so there's no need to press RETURN until the lines are full—although you can enter shorter strings if you want to. And the program can hold a total of 256 of these strings.

When you first start to enter a string of notes, the notes are printed on the screen but there's no sound. If you want to practise the tune without storing the notes press CTRL and ←. Press P (then delete the P from the screen) to print the notes as you play them and Q to cancel the print—so you're back to play only again. Pressing RETURN enters the string into memory and starts you off with a new string and the print only mode.

Tempo, octave, voice and volume can be changed at any time in the tune simply by entering their letter and new value in the appropriate place in the string of notes.

To replay the tune simply type PLAY, but if you want to see the notes printed out at the same time press ? instead. If you do choose to see the printout you can pause the tune while it is being replayed by pressing *. If you don't want to replay the entire tune you can choose the starting point by pressing ← and then entering the start note number.

To edit the tune use the asterisk to stop the listing at the string you want to edit then use the Vic's normal editor to change, add or delete notes. The only thing to watch for when adding notes is that the string doesn't extend beyond the maximum of four lines.

The Acorn program is in two parts. Type in and SAVE the first part which sets up the UDGs and ENVELOPEs, then type in and SAVE the second part using the name COMP. When you RUN the Acorn program the first part will automatically CHAIN the second part. You will be presented with three musical staves and, at the top of the screen, pictures of all the notes from a semi-breve to a semi-quaver, a selection of rests and a treble and bass clef. The program is controlled from the keyboard rather than a menu. The keys which do something are: L-Load a tune; S-Save a tune; P-Play tune; C-Change tempo; D-Delete note; I-Insert note; Q-Finish inserting; T-Tie to next note; E-Change envelope; and V-Display section of tune.

To create a piece of music you first have to select a clef. Do this by moving the rectangle to the correct clef using the left and right cursor keys, and then press RETURN. The clef will appear at the left of the first stave. You can now start selecting the notes and rests to build up the music. The method is the same as before—move the box to the note then press RETURN. But this time, instead of the note itself, a circle appears on the stave which you can move to the correct position using the up and down cursor keys. When you press RETURN a second time the note you selected sounds and appears on the stave. If you want a sharp or flat, position this first and then the note. Notes can be positioned above or below the stave as any short ledger lines needed will appear automatically. Rests are inserted in the same way as notes. The program even allows you to tie or slur two notes together. Position the first note as usual, press T, then position the second note (this must be on the same line) and the tie will appear.

When you've entered a few notes, press P to play the tune. You'll be asked for the start and end note numbers, but if you want to hear the whole tune just press RETURN twice.

Now try changing the tempo by pressing C. You'll hear a regular drum beat which you can speed up or slow down by pressing the up or down cursor keys. Keep your finger on the key until the tempo is just right.

At some stage you're sure to want to edit the tune. To delete a note or series of notes first work out the number of the notes you want to remove—the numbers of the first note in each stave is displayed above it. Now press D, and enter the first and last number of the series or one number for a single wrong note. Press RETURN and the notes are deleted.

To insert notes, press I then enter the number of the note *before* the notes to be inserted. Then select and position the notes as normal—they will appear in the correct position. When you've finished, press Q.

The V key is used to display any section of the tune. Enter the number of the start note and two staves full of notes will be displayed from that position.

The last command, E, lets you change the sound of the note. Five envelopes have been



set up for you to choose from—each of the pre-programmed sounds mimics different instruments. These are organ, vibrato 1, vibrato 2, harpsichord and piano, and to select them just type in a number from 1 to 5.

When you first RUN the program you'll be presented with a main menu offering nine options. Start with option 3 to play music directly on the keyboard. The notes on the keyboard are laid out in two rows in the same way as for the music article on pages 669 to 675 with Q as bottom C, I as middle C and V as top C. The computer remembers each note you play, but not in real time, so it doesn't matter if you hesitate between notes, and if you play a wrong note it is very easy to edit later on.

If you want to change the octave before you play a note, use the up or down arrow keys. If you want to change the length of the note use the left and right arrow keys. The current octave and length of note are printed at the top of the screen while you're playing and the notes are also printed as they are entered.

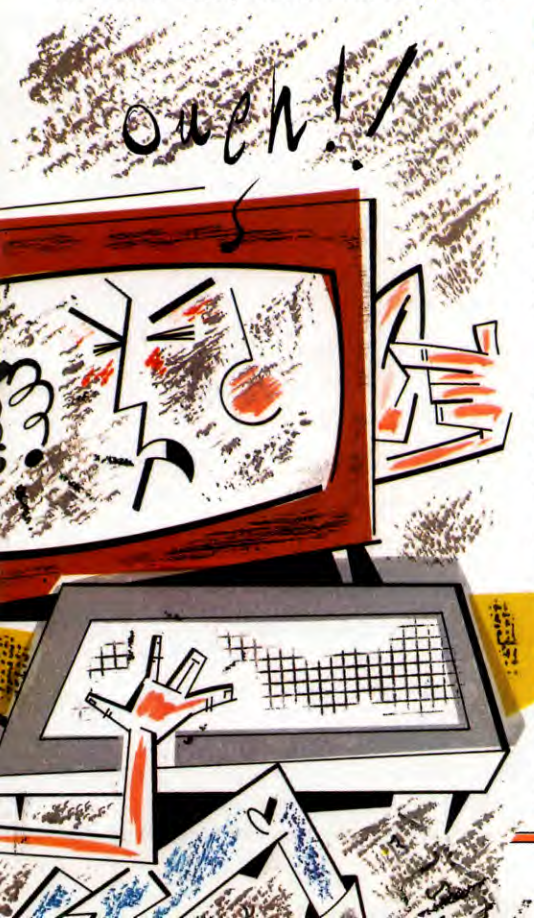
When you're happy with this method of entering the notes, go on to option 4 (the notes you've played will remain in memory). This gives you another way of entering the notes which is also quite straightforward. First type in the name of the note, from A to F for natural notes, shifted A C D F G for sharps (flats are entered as the sharp below, so B♭ = A♯ etc.) and P for a pause. Now enter the octave, from 1 to 5 and then the length of the note, using the initial letter of whole, half, quarter, eighth or sixteenth. For a dotted note simply add a dot at the end. For example, A2e. means A sharp, octave 2, dotted quaver; C3w means note C, octave 3, semi-breve.

Enter as many notes as you like. You can hear the tune at any time by choosing option 7 from the main menu.

Other options from the menu which effect the playback are those to change the tempo and change the octave—simply type in the new value. Tempo can range from Ø to 255 and the octave is raised or lowered one octave at a time by typing U for up or D for down.

At some time you're sure to need to edit the tune, so choose option 6 and practise on your sample tune. First use this to list the notes and write down the numbers of the notes you want to change. You're then given a choice of delete, insert, change or continue. Try each one in turn. To delete some notes enter the start number and the number of notes you want to delete. To insert, enter the number of the note *before* the point you want to insert and then type in the notes in the usual format. To change a note simply input its number then type in the new contents.

Finally, when you're happy with the tune, you can save the music with option 2 and load it back in again with option 1.




```
4452 INPUT "Enter Number of Note
   – □□";NN
4454 IF (NN<1) OR NN>ct THEN GOTO
   4000
4460 PRINT : PRINT "Re-entering Note□";NN
4470 PRINT : PRINT
4480 INPUT "Enter New Note – □□";N$
4490 IF N$="" THEN GOTO 4300
4500 FOR i=1 TO LEN (N$): IF (N$(i)<"Ø"
   OR N$(i)>"9") AND (N$(i)<>"–")
   THEN GOTO 4000
4510 LET N=VAL (N$)
4520 IF INT (N/1000)>11 THEN GOTO 4000
4530 IF N<Ø THEN GOTO 4590
4540 LET M=INT (N/100): LET
   D=N–M*100
4550 LET O=M–INT (M/10)*10: IF O<1
   OR O>7 THEN GOTO 4000
4560 LET M=INT (M/10)+(O–1)*
   12–36
4570 LET t(2*NN–1)=D: LET t(2*NN)=M
4580 GOTO 4000
4590 LET M=INT (N/100)+1: LET
   D=Ø–(N–M*100)
4600 IF M<>–4 THEN GOTO 4000
4610 GOTO 4570
4700 CLS
4705 PRINT "Enter Number of Note
   BEFORE"'" new note to be
   inserted."'"(Enter Ø to Exit)"
4730 INPUT is
4735 IF is=Ø THEN GOTO 4000
4740 IF is>ct+1 THEN GOTO 4700
4745 CLS
4750 GOSUB 2500
4755 PRINT
4760 INPUT "Enter New Note – □□";N$
4765 IF LEN (N$)=Ø THEN GOTO 4700
4770 FOR i=1 TO LEN (N$): IF (N$(i)<"Ø"
   OR N$(i)>"9") AND (N$(i)<>"–")
   THEN GOTO 4700
4772 LET N=VAL (N$)
4774 IF INT (N/1000)>11 THEN GOTO 4700
4776 IF N<Ø THEN GOTO 4792
4778 LET M=INT (N/100): LET
   D=N–M*100
4782 LET O=M–INT (M/10)*10: IF O<1
   OR O>7 THEN GOTO 4700
4784 LET M=INT (M/10)+(O–1)*
   12–36
4786 FOR i=ct TO is STEP –1: LET
   t(2*(i+1))=t(2*i): LET
   t(2*(i+1)–1)=t(2*i–1): NEXT i
4788 LET t(2*is–1)=D: LET t(2*is)=M
4790 LET ct=ct+1: GOTO 4000
4792 LET M=INT (N/100)+1: LET
   D=Ø–(N–M*100)
4794 IF M<>–4 THEN GOTO 4700
4796 GOTO 4786
4800 CLS
4805 PRINT "Enter number of note to
   be"'"deleted."'"(Enter Ø to Exit)"'
4830 INPUT de
4835 IF de=Ø THEN GOTO 4000
4840 IF de>ct THEN GOTO 4800
4845 FOR i=de TO ct: LET t(2*i)=t(2*
   (i+1)): LET t(2*i–1)=t(2*(i+1)–1):
   NEXT i
4850 LET ct=ct–1
4855 GOTO 4000
4900 DATA 3,Ø,–11,–9,Ø,–6,–4,–2,Ø,1
4910 DATA 12,9,8,–8,10,Ø,13,Ø,15,Ø
4920 DATA Ø,16,14,2,4,–12,–7,6,–5,–1
4930 DATA 11,–10,7,–3,5
5000 CLS
5010 INPUT "ENTER FILENAME ?";F$: LET
   T(maxnotes+1)=CT: SAVE F$ DATA T():
   RETURN
6000 INPUT "ENTER FILENAME ?";F$: LOAD
   F$ DATA T(): LET CT=T(maxnotes+1):
   RETURN
```

```
5670 LP=Ø
5680 GOSUB 6000:RETURN
5690 IF BP THEN 5710
5700 BP=1:GOTO 5720
```

```
5710 BP=0
5720 GOSUB 6000:RETURN
5730 IF HP THEN 5750
5740 HP=1:GOTO 5760
5750 HP=0
5760 GOSUB 6000:RETURN
5770 IF V3 THEN 5790
5780 V3=1:GOTO 5800
5790 V3=0
5800 GOSUB 6000:RETURN
5810 GOSUB 5900
5820 PRINT"ENTER VALUE — □□";
5830 INPUT V
5840 IF V<0 OR V>20 THEN RETURN
5850 TEMPO=200-10*V
5860 RETURN
5900 PRINT "■"
5910 FOR I=1 TO 15:PRINT "■";:NEXT I
5920 RETURN
6000 FOR I=0 TO 2
6010 F2=INT(FR(I)/256)
6020 F1=FR(I)-(F2*256)
6030 POKE SID+(I*7),F1:
   POKE SID+(I*7)+1,F2
6040 F2=INT(PW(I)/256)
6050 F1=PW(I)-(F2*256)
6060 POKE SID+(I*7)+2,F1:
   POKE SID+(I*7)+3,F2
6070 F1=AT(I)*16+DE(I)
6080 POKE SID+(I*7)+5,F1
6090 F1=SU(I)*16+RE(I)
6100 POKE SID+(I*7)+6,F1
6110 CR(I)=SY(I)*2+RM(I)*4+
   (16*2↑(WF(I)))
6115 NEXT I
6120 F2=INT(FF/8)
6130 F1=FF-(F2*8)
6140 POKE SID+21,F1:
   POKE SID+22,F2
6150 F1=FR*16+FI(0)+FI(1)*2+FI(2)*4
6160 POKE SID+23,F1
6170 F1=VOL+LP*16+BP*32+HP*64+
   V3*128
6180 POKE SID+24,F1
6190 RETURN
7000 GOSUB 9100
7050 PRINT NS;"□NOTES ALREADY
   ENTERED (1500 MAX.)■■"
7065 N$=""
7070 INPUT N$:IF LEN(N$)=0 THEN
   RETURN
7080 N=VAL(N$)
7082 IF INT(N/100)>11 THEN 7000
7085 IF N<0 THEN 7120
7090 M=INT(N/10):V=N-M*10:
   V=V-1:IF V<0 OR V>2 THEN 7000
7100 VOICE(V,NS(V))=M:NS=NS+1:
   NS(V)=NS(V)+1
7110 GOTO 7000
7120 N=0-N:M=INT(N/10)
7130 V=N-M*10:V=V-1:IF V<0 OR
   V>2 THEN 7000
7140 M=0-M:GOTO 7100
7500 PRINT "♥TUNE EDITOR"
7520 PRINT"■■■C — CLEAR ALL
   NOTES"
7530 PRINT"■D — DISPLAY ALL NOTES
   FOR A VOICE"
7540 PRINT"■E — EDIT/DELETE A NOTE"
7544 PRINT"■I — INSERT A NOTE"
7550 PRINT"■R — RETURN TO MAIN
   MENU"
7600 GET E$:IF E$="" THEN 7600
7610 IF E$="R" THEN RETURN
7620 IF E$="C" THEN 7800
7630 IF E$="D" THEN 7900
7640 IF E$="E" THEN 7700
7650 IF E$="I" THEN 7850
7660 GOTO 7600
7700 PRINT"♥■■■ENTER VOICE
   NUMBER —□□";
7730 INPUT V
7735 IF V<1 OR V>3 THEN 7700
7740 PRINT"■ENTER NOTE NUMBER
   —□□";
7745 INPUT NO
7750 IF NO<1 OR NO>NS(V-1) THEN
   7500
7755 PRINT"■■NOTE IS CURRENTLY —
   □□";VOICE(V-1,NO-1)
7765 PRINT"■ENTER NEW VALUE — (0 TO
   DELETE)□□";
7770 INPUT VN
7772 IF VN=0 THEN 7791
7773 IF VN=-2 OR VN=-1 THEN 7780
7775 IF INT(VN/10)>11 OR VN-(INT(VN/
   10)*10)>7 THEN 7765
7780 VOICE(V-1,NO-1)=VN
7790 GOTO 7500
7791 PRINT"■>■■PLEASE WAIT"
7792 FOR I=NO-1 TO 499
7794 VOICE(V-1,I)=VOICE(V-1,I+1)
7796 NEXT I
7797 NS(V-1)=NS(V-1)-1
7798 GOTO 7500
7800 PRINT"■■■PLEASE WAIT FOR
   ARRAY TO BE ZEROED"
7810 GOSUB 2050:NS=0:FOR I=0 TO
   2:NS(I)=0:NEXT I:GOTO 7500
7850 PRINT"♥■■■ENTER NUMBER OF
   NOTE IN THE REQUIRED"
7855 PRINT"VOICE, BEFORE THE NOTE TO BE
   INSERTED"
7860 PRINT"■—□";
7862 INPUT NO
7864 IF NO<1 THEN 7500
7865 NO=NO-1
7866 GOSUB 9100
7868 N$=""
7870 INPUT N$:IF LEN(N$)=0 THEN 7500
7872 N=VAL(N$)
7874 IF INT(N/100)>11 THEN 7500
7876 IF N<0 THEN 7890
7878 M=INT(N/10):V=N-M*10:V=V-1:
   IF V<0 OR V>2 THEN 7500
7880 FOR I=NS(V) TO NO STEP -1
7882 VOICE(V,I+1)=VOICE(V,I)
7884 NEXT I
7886 VOICE(V,NO)=M:NS=NS+1:NS(V)=
   NS(V)+1
7888 GOTO 7500
7890 N=0-N: M=INT(N/10)
7892 V=N-M*10:V=V-1:IF V<0 OR
   V>2 THEN 7500
7894 M=0-M:GOTO 7880
7900 PRINT"■■ENTER VOICE NUMBER
   —□";
7920 INPUT V:IF V<1 OR V>3 THEN 7500
7925 PRINT "♥"
7930 FOR I=1 TO NS(V-1)+1
7940 PRINT I,
```



```
7942 PRINT INT(VOICE(V−1,I−1)/1Ø),
7944 PRINT VOICE(V−1,I−1)−INT(VOICE
    (V−1,I−1)/1Ø)*1Ø
795Ø IF I=15*INT(I/15) THEN 796Ø
7955 GOTO 798Ø
796Ø PRINT"■HIT ANY KEY TO CONTINUE"
797Ø GET Z$:IF Z$="" THEN 797Ø
798Ø NEXT I
7982 PRINT"■■HIT ANY KEY TO
    RETURN"
7985 GET Z$:IF Z$="" THEN 7985
799Ø GOTO 75ØØ
8ØØØ DATA 1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,
    2
8Ø1Ø DATA 3,3,3,3,3,4,4,4,4,5,5,5,6,6,7,7,7,8,
    8
8Ø2Ø DATA 9,9,1Ø,11,11,12,13,14,14,15,16,
    17,18
8Ø3Ø DATA 19,21,22,23,25,26,28,29,31,33,35,
    37
8Ø4Ø DATA 39,42,44,47,5Ø,53,56,59,63,67,71,
    75
8Ø5Ø DATA 79,84,89,94,1ØØ,1Ø6,112,119,126,
    134
8Ø6Ø DATA 142,15Ø,159,168,179,189,2ØØ,
    212,225
8Ø7Ø DATA 238,253,12,28,45,62,81,1Ø2,123,
    145,169
8Ø8Ø DATA 195,221,25Ø,24,56,9Ø,125,163,
    2Ø4,246
8Ø9Ø DATA 35,83,134,187,244,48,112,18Ø,
    251,71,152,237,71
81ØØ DATA 167,12,119,233,97,225,1Ø4,247,
    143,48
811Ø DATA 218,143,78,24,239,21Ø,195,195,
    2Ø9,239
812Ø DATA 31,96,181,3Ø,156,49,223,165,135,
    134
813Ø DATA 162,223,62,193,1Ø7,6Ø,57,99,19Ø,
    75
814Ø DATA 15,12,69,191,125,131,214,121,115
815Ø DATA 199,124,151,3Ø,24,139,126,25Ø,6,
    172,243,23Ø,143,248,46
9ØØØ FOR I=Ø TO 24:POKE SID+I,Ø:NEXT
    I:RETURN
91ØØ PRINT "■ENTER NOTES AS
    <NUMBER>,<OCTAVE>,<VOICE>
    ■■"
912Ø PRINT"C−ØE−4G#−8"
913Ø PRINT"C#−1F−5A−9"
914Ø PRINT"D−2F#−6A#−1Ø"
915Ø PRINT"D#−3G−7B−11"
917Ø PRINT"■REST−−1SUSTAIN−−2"
918Ø PRINT"■OCTAVE − Ø TO 7"
919Ø PRINT"■E.G. □□ 1Ø31 <RET> −
    A FLAT, OCT.3 VOICE 1"
92ØØ PRINT"■■□□□□□□□ØØ42
    <RET> − C, OCT.4 VOICE 2":RETURN
93ØØ INPUT "■ENTER NAME TO
    SAVE";NM$:OPEN 1,1,1,NM$
931Ø C$=",":FOR Z=Ø TO 5ØØ
932Ø FOR ZZ=Ø TO 2:PRINT#1,
    VOICE(ZZ,Z)
933Ø IF VOICE(Ø,Z)<>Ø OR VOICE
    (1,Z)<>Ø OR VOICE(2,Z)<>Ø THEN
    NEXT ZZ,Z
9331 FOR Z=Ø TO 2:PRINT#1,WF$(Z)C$AT
    (Z)C$DE(Z)C$SU(Z)C$RE(Z)C$CR(Z)
9332 PRINT#1,PW(Z)C$FR(Z)C$SY(Z)C$RM
    (Z)C$FI(Z)C$NS(Z):NEXT Z:PRINT#1,NS
934Ø CLOSE1:RETURN
94ØØ INPUT "■ENTER NAME TO
    LOAD";NM$:OPEN 1,1,Ø,NM$
941Ø FOR Z=Ø TO 5ØØ
942Ø FOR ZZ=Ø TO 2:INPUT#1,
    VOICE(ZZ,Z)
943Ø IF VOICE(Ø,Z)<>Ø OR VOICE
    (1,Z)<>Ø OR VOICE(2,Z)<>Ø THEN
    NEXT ZZ,Z
9431 FOR Z=Ø TO 2:INPUT#1,WF$(Z),AT
    (Z),DE(Z),SU(Z),RE(Z),CR(Z)
9432 INPUT#1,PW(Z),FR(Z),SY(Z),RM(Z),FI
    (Z),NS(Z):NEXT Z:INPUT#1,NS
944Ø CLOSE1:RETURN
```

```
1Ø DIM A$(255):POKE 36879,14:X=Ø
5ØØ GOSUB 1ØØØ:PRINT "■■■";:
    A$=A$(X):INPUT A$
5Ø5 IF LEFT$(A$,1)="←" THEN INPUT
    "■■■■■■■■ENTER PLAY";
    X:X=X AND 255:GOTO 5ØØ
51Ø IF LEFT$(A$,4)="PLAY" THEN 2ØØØ
511 IF LEFT$(A$,5)="STORE" THEN 3ØØØ
512 IF LEFT$(A$,1)="↑" THEN
    A$(X)="":GOTO 5ØØ
513 IF LEFT$(A$,1)="?" THEN 2Ø1Ø
515 IF LEFT$(A$,3)="CLR" THEN RUN
52Ø IF LEFT$(A$,3)="END" THEN POKE
36879,27:PRINT "■■":END
55Ø A$(X)=A$:X=X+1:GOTO 5ØØ
1ØØØ IF X<Ø THEN X=255
1ØØ1 IF X>255 THEN X=Ø
1ØØ2 POKE 36867,138:PRINT "■"TAB(13)
    "■PLAY"X:PRINT "■■■"A$(X)
1ØØ5 PRINT "■■■■■■■■
    MAKING MUSIC WITH■THE SUPER
    EXPANDER■";
1Ø1Ø PRINT "■VOLUME:V+(Ø−9)"
1Ø15 PRINT "■VOICE:S+(1−4)"
1Ø2Ø PRINT "■OCTAVE:O+(1−3)"
1Ø25 PRINT "■TEMPO:T+(Ø−9)"
1Ø3Ø PRINT "■NOTES:
    ■C■D■E■F■G■A■B"
1Ø35 PRINT "■REST:R■[TEMPO]"
1Ø4Ø PRINT "■SHARPS:#+(NOTE)"
1Ø45 PRINT "■FLATS:$+(NOTE)"
1Ø5Ø PRINT "■MOVEPLAY:□(←)"
1Ø54 PRINT '■PLAYTUNE:□(PLAY),(?)"
1Ø55 PRINT "■■(CTRL)+(←)−PLAY
    MODE□□";
1Ø56 PRINT "■(P)−PRINT IN PLAY
    MODE";
1Ø57 PRINT "(Q)−CANCEL PRINT
    □□□□□□■";
1Ø58 PRINT"■(CLR)−CLEAR PLAY
    □□□□□□□";
1Ø59 PRINT"(END)−QUIT PROGRAM
    □□□□";
1Ø6Ø PRINT "(STORE)−SAVE AND
    LOAD■";
1Ø65 POKE 8185,16Ø:POKE 389Ø5,6
1Ø7Ø POKE 36867,46:RETURN
2ØØØ POKE 36867,138:PRINT"■■■■>
    ■■PLAYING TUNE"
2ØØ3 FOR Z=Ø TO 255:IF A$(Z)<>""
    THEN PRINT "■Q"+A$(Z)
2ØØ5 NEXT Z:GOTO 5ØØ
2Ø1Ø PRINT"■":FOR Z=Ø TO 255:PRINT
    "■■"Z"■";"■■P■"+A$(Z)
2Ø15 GET Z$:IF Z$=CHR$(13) THEN
    Z=255:GOTO 2ØØ5
2Ø16 IF Z$="*" THEN POKE 198,Ø:WAIT
    198,1:POKE198,Ø
2Ø2Ø NEXT Z:INPUT "PRESS RETURN";Z$:
    GOTO 5ØØ
3ØØØ N$="":INPUT "■■NAME";N$
3Ø1Ø PRINT "■■S■AVE OR ■L■OAD
    TO TAPE?■"
3Ø2Ø GET Z$:IF Z$<>"S" AND
    Z$<>"L" THEN 3Ø2Ø
3Ø3Ø IF Z$="S" THEN 3Ø5Ø
3Ø4Ø OPEN 1,1,Ø,N$:FOR Z=Ø TO 255:
    INPUT#1,A$(Z):NEXT Z:CLOSE1:
    GOTO 5ØØ
3Ø5Ø OPEN 1,1,1,N$:FOR Z=Ø TO 255:
    PRINT#1,CHR$(34)A$(Z)CHR$(34):
    NEXT Z:CLOSE1:GOTO 5ØØ
```

```
195Ø DEFPROCDrawbox
196Ø MOVEI%*8Ø+9Ø,64Ø:DRAWI%*8Ø+9Ø,
    78Ø:DRAWI%*8Ø+17Ø,78Ø:DRAWI%*
    8Ø+17Ø,64Ø:DRAWI%*8Ø+9Ø,64Ø:
    ENDPROC
197Ø DEFPROCPlaytune
198Ø VDU4:CLS
199Ø PRINTTAB(5,2)"Enter the starting
    note.□";
2ØØØ PROCRead
2Ø1Ø IFD%>=NO%PROCrange(1):
    GOTO198Ø
2Ø2Ø L%=D%:CLS
2Ø3Ø PRINTTAB(5,2)"Enter the ending
    note.□";:PROCRead
2Ø4Ø IFD$=""R%=NO%−1ELSER%=D%
2Ø5Ø IFR%>=NO%PROCrange(1):GOTO
    2Ø3Ø
2Ø6Ø IFR%<L%PROCrange(Ø):
    GOTO2Ø3Ø
2Ø7Ø CLS:VDU5:C%=L%:IFC%=ØTHEN
    21ØØ
2Ø8Ø REPEAT:C%=C%−1
```

```
2Ø9Ø UNTILS(Ø,C%)=256ORS(Ø,C%)=257
2100 IFS(Ø,C%)=256THENF2%=ØELSE
   F2%=-1
211Ø X=L%-1:REPEAT:X=X+1
212Ø D%=INKEY(3)
213Ø IFD%<>-1SOUND17,Ø,Ø,Ø:X=R%:
   GOTO224Ø
214Ø IFX>R%THEN224Ø
215Ø IFS(Ø,X)=256F2%=Ø:X=X+1:GOTO
   214Ø
216Ø IFS(Ø,X)=257F2%=-1:X=X+1:
   GOTO 214Ø
217Ø IFS(Ø,X)>257SOUND1,Ø,Ø,INT
   ((2∧(11-S(1,X))*SD)+.5):GOTO224Ø
218Ø IFF2%B%=S(Ø,X)ELSEB%=S(Ø,X)+
   12
219Ø C%=Ø
22ØØ IFS(Ø,X)-INT(S(Ø,X))=.5THEN
   C%=-4
221Ø IFS(Ø,X)-INT(S(Ø,X))=.75THEN
   C%=4
222Ø SOUND1,EV%,PT%(B%)+C%,INT((D1%
   (S(1,X)-1)*SD)+.5)
223Ø SOUND1,Ø,Ø,Ø
224Ø UNTIL X>=R%
225Ø PROCno:ENDPROC
226Ø DEFPROCE
227Ø G%=(I%=5)
228Ø I%=Ø
229Ø PROCM(Ø,4)
23ØØ PROCDrawnote(D%(I%),SE%+NS(NE))
231Ø B%=NE-(F%=Ø)*12
232Ø IFNOTG%S(Ø,NO%)=NE+.5:SOUND1,
   EV%,PT%(B%)-4,8
233Ø IFG%S(Ø,NO%)=NE+.75:SOUND1,
   EV%,PT%(B%)+4,8
234Ø S(1,NO%)=D%(I%):ENDPROC
235Ø DEFPROCM(L%,R%)
2355 *FXI5,Ø
236Ø GCOL4,1:PROCDrawbox:REPEAT
237Ø REPEAT:A%=GET
238Ø UNTILA%=136ORA%=137ORA%=13
   ORA%=67ORA%=68ORA%=69OR
   A%=73ORA%=76ORA%=79ORA%=
   8ØORA%=81ORA%=83ORA%=84OR
   A%=86
239Ø IFA%<136GOTO246Ø
24ØØ PROCDrawbox
241Ø IFA%=136I%=I%-1
242Ø IFA%=137I%=I%+1
243Ø IFI%<L%I%=R%
244Ø IF I%>R%I%=L%
245Ø PROCDrawbox
246Ø UNTIL A%<136
247Ø PROCDrawbox:GCOLØ,1:ENDPROC
248Ø DEFPROCEnter
249Ø GCOL4,1
25ØØ PROCDrawnote(D%(Ø),SE%+NS(NE)):
   X%=X%-S%
251Ø REPEAT:REPEAT:A%=GET
252Ø UNTIL(A%=138ANDNE<>Ø)OR
   (A%=139ANDNE<17)ORA%=13OR
   A%=67ORA%=68ORA%=69ORA%=73
   ORA%=76ORA%=79ORA%=8ØOR
   A%=83ORA%=84ORA%=86
253Ø IFA%>13ANDA%<138GOTO258Ø
254Ø PROCDrawnote(D%(Ø),SE%+NS(NE)):
   X%=X%-S%
255Ø IFA%=138NE=NE-1
256Ø IFA%=139NE=NE+1
257Ø PROCDrawnote(D%(Ø),SE%+NS(NE)):
   X%=X%-S%
258Ø UNTILA%<138
259Ø PROCDrawnote(D%(Ø),SE%+NS(NE)):
   X%=X%-S%
26ØØ IFA%>13ENDPROC
261Ø GCOLØ,1:D%=SE%+NS(NE)
262Ø IFI%<5PROCDrawnote(D%(I%),D%)
263Ø IFI%=5PROCACC(S$,D%):PROCE:
   ENDPROC
264Ø IFI%=6PROCACC(F$,D%+2):PROCE:
   ENDPROC
265Ø B%=NE-12*(F%=Ø)
266Ø S(Ø,NO%)=NE:S(1,NO%)=D%(I%)
267Ø SOUND1,EV%,PT%(B%),8:ENDPROC
268Ø DEFPROCDisplay(L%,R%)
269Ø VDU5,12:PROCno:VDU5
27ØØ GCOLØ,1:IFL%<1THENL%=1
271Ø E%(Ø)=L%:NE=9
272Ø PROCDisplaysymbols
273Ø PROCDrawthreestaves
274Ø X%=2Ø:MOVE65,ST%(Ø)+14Ø:PRINT;
   L%
275Ø SE%=ST%(Ø):H%=Ø:C%=L%:
   REPEAT
276Ø IFX%<>2Ø GOTO 281Ø
277Ø D%=C%+1:REPEAT:D%=D%-1
278Ø UNTILS(Ø,D%)=256ORS(Ø,D%)=257
279Ø IFS(Ø,D%)=256PROCDrawtrebleclef
   (SE%)ELSEPROCDrawbassclef(SE%)
28ØØ IFNO%=1THEN291Ø
281Ø IFS(Ø,C%)=256THENPROCDrawtreble
   clef(SE%)
282Ø IFS(Ø,C%)=257PROCDrawbassclef
   (SE%)
283Ø IFS(Ø,C%)>257PROCDrawrest(S(Ø,
   C%)-258,SE%+64)
284Ø IFS(Ø,C%)-INT(S(Ø,C%))=.5PROC
   ACC(F$,SE%+NS(S(Ø,C%)))
285Ø IFS(Ø,C%)-INT(S(Ø,C%))=.75PROC
   ACC(S$,SE%+NS(S(Ø,C%)))
286Ø IFS(Ø,C%)<2ØØNE=INT(S(Ø,C%)):
   PROCDrawnote(S(1,C%),SE%+NS(S(Ø,
   C%)))
287Ø IFX%<=11ØØGOTO291Ø
288Ø H%=H%+1:X%=2Ø:SE%=ST%(H%):
   E%(H%)=C%+1
289Ø MOVE65,SE%+14Ø
29ØØ IFH%=2PRINT;NO%ELSEPRINT;C%+1
291Ø C%=C%+1:UNTILH%=2ORNO%=1
   ORC%=R%
292Ø IFX%=2ØANDF%PROCDrawbassclef
   (SE%)
293Ø IFX%=2ØANDNOTF%PROCDrawtreble
   clef(SE%)
294Ø NE=INT(NE):*FX15,1
295Ø ENDPROC
296Ø DEFPROCInit
297Ø TLT%=6ØØ:LT%=TLT%:EV%=1:
   SD=2:NO%=1
298Ø C%=1:I%=12:S%=6Ø:H%=Ø:
   F%=Ø:F3%=Ø
299Ø DIMT$(3),B$(1),R$(5,1),ST%(2),
   NS(17),S(1,LT%),PT%(29),E%(2),D%(4),
   D1%(16),X%(2Ø),Y%(2Ø)
3ØØØ ST%(Ø)=5ØØ:ST%(1)=3ØØ:ST%(2)=
   1ØØ
3Ø1Ø SE%=ST%(Ø): NE=9
3Ø2Ø E%(Ø)=1:E%(1)=Ø:E%(2)=Ø
3Ø3Ø T$(Ø)=CHR$(255)+CHR$(254)
3Ø4Ø T$(1)=CHR$(253)+CHR$(252)
3Ø5Ø T$(2)=CHR$(251)+CHR$(25Ø)
3Ø6Ø T$(3)=CHR$(249)
3Ø7Ø B$(Ø)=CHR$(248)+CHR$(247)
3Ø8Ø B$(1)=CHR$(246)+CHR$(245)
3Ø9Ø N1$=CHR$(243):N2$=CHR$(242)
31ØØ S$=CHR$(231):F$="b"
311Ø FORD%=241TO232STEP-2
312Ø R$(((241-D%)/2),Ø)=CHR$(D%)
313Ø R$(((241-D%)/2),1)=CHR$(D%-1)
314Ø NEXT
315Ø VDU19,Ø,7;Ø;19,1,Ø;Ø;28,Ø,7,39,Ø:
   *FX4,1
316Ø FORX=ØTO4:READD%(X):NEXT
317Ø FORX=ØTO15:READD1%(X):NEXT
318Ø FORX=ØTO17:READNS(X):NEXT
319Ø FORX=ØTO29:READPT%(X):NEXT
32ØØ FORX=ØTO2Ø
321Ø X%(X)=INT((COS(X/2Ø*PI)*-3Ø)+
   .5):Y%(X)=INT((SIN(X/2Ø*PI)*-13)+.5)
322Ø NEXT:VDU5
323Ø PROCDisplaysymbols
324Ø PROCDrawthreestaves
325Ø X%=2Ø:REPEAT:PROCM(12,13)
326Ø UNTILA%=13ORA%=76
327Ø IFA%=76PROCLoad:GOTO331Ø
328Ø IFI%=12PROCDrawtrebleclef(SE%):S(Ø,
   Ø)=256:F%=Ø
329Ø IFI%=13PROCDrawbassclef(SE%):
   S(Ø,Ø)=257:F%=-1
33ØØ PROCno:MOVE1ØØ,64Ø:VDU5,49
331Ø I%=Ø:ENDPROC
332Ø DEFPROCno
333Ø VDU4,12:IFNO%=LT%+1PRINTTAB
   (7,2)"There is no more room left":
   ENDPROC
334Ø PRINTTAB(15,6)"No:";NO%:ENDPROC
335Ø DEFPROCrange(X)
336Ø VDU7,12,31,9,2:IFX=1THENPRINT
   "Number too large."ELSEPRINT"Number
   too small."
337Ø TIME=Ø:REPEATUNTILTIME>15Ø:
   ENDPROC
```

```
1440 FOR I=ST TO NN
1450 PRINT#C,USING "###□□";I;
1460 A$=N$(I):B$=LEFT$(A$,1)
1470 IF B$>="a" AND B$<="g" THEN
   C$=CHR$(ASC(B$)-32)+"□" ELSE
   C$=B$+"#"
1480 IF B$="p" THEN C$="-□"
1490 PRINT#C,C$;
1500 IF B$<>"p" THEN PRINT#C,"□□
   OCTAVE #";MID$(A$,2,1); ELSE PRINT
   #C,STRING$(11,32);
1510 PRINT#C,"□□";LE$(INSTR(R1$,
   MID$(A$,3,1)));
1520 IF MID$(A$,4,1)="." THEN
   PRINT#C,"." ELSE PRINT#C
1530 LP=LP+1:IF LP=13 AND C=Ø THEN
   LP=Ø:GOSUB94Ø:CLS
1540 NEXT:GOSUB94Ø
1550 PRINT@448,STRING$(63,32);:PRINT
   @48Ø,"PRESS ANY KEY FOR MENU";:
   EXEC 41194:RETURN
1560 CLS
1570 PRINT@8,"PLAY NOTES MODE"
1580 IF NN=Ø THEN RETURN
1590 PRINT:INPUT "START AT NOTE?□
   (ENTER=1)";ST
1600 IF ST<=Ø THEN ST=1 ELSE IF
   ST>NN THEN ST=NN
1610 PRINT"TEMPO =";TE
1620 PLAY "V31;T"+STR$(TE)
1630 EXEC 36Ø55:FOR I=ST TO NN
1640 PRINT@256,"PLAYING NOTE
   NUMBER";I
1650 A$=N$(I)
1660 B$=LEFT$(A$,1):IF B$="p" THEN
   169Ø ELSEIF B$>="a" AND
   B$<="g" THEN
   P$=CHR$(ASC(B$)-32)+";" ELSE
   P$=B$+"#"
1670 PLAY "O"+MID$(A$,2,1)+"L"+L2$
   (INSTR(R1$,MID$(A$,3,1)))+MID$(A$,
   4,1)+P$
1680 GOTO 17ØØ
1690 PLAY "P"+L2$(INSTR(R1$,MID$(A$,
   3,1)))
1700 NEXTI
1710 RETURN
1720 CLS
1730 IF NN=Ø THEN RETURN
1740 PRINT@5,"GLOBAL OCTAVE CHANGE"
1750 PRINT@64,"OCTAVE SHIFT UP OR
   DOWN (U/D)"
1760 POKE 329,255:INPUT A$
1770 IF A$="" THEN RETURN
1780 IF A$<>"U" AND A$<>"D" THEN
   175Ø
1790 PRINT"START AT (ENTER=ALL)";:
   INPUT ST
1800 IF ST<=Ø THEN ST=1:EN=NN:
   GOTO 184Ø
1810 INPUT "END AT (ENTER=
   END)";EN
1820 IF EN=Ø OR EN>NN THEN EN=NN
1830 IF ST>EN THEN ST=EN
1840 FOR I=1 TO NN
1850 B$=MID$(N$(I),2,1)
1860 IF A$="D" THEN C$=CHR$(ASC
   (B$)-1):IF C$="Ø" THEN C$="5"
1870 IF A$="U" THEN C$=CHR$(ASC
   (B$)+1):IF C$="6" THEN C$="1"
1880 MID$(N$(I),2,1)=C$
1890 NEXT:RETURN
1900 CLS
1910 PRINT@6,"LOAD MUSIC FROM TAPE"
1920 PRINT:PRINT"THIS OPTION WILL ERASE
   ANY MUSICIN MEMORY - DO YOU
   WANT TO GO□□□AHEAD (Y/N)";
1930 POKE 329,255:INPUT A$
1940 IF A$<>"Y" THEN RETURN
1950 PRINT:LINE INPUT "FILENAME:";A$
1960 OPEN "I",#-1,A$
1970 INPUT #-1,N$,T$
1980 NN=VAL(N$):TE=VAL(T$)
1990 FOR I=1 TO NN
2000 INPUT #-1,N$(I)
2010 NEXT:CLOSE#-1:RETURN
2020 CLS
2030 IF NN=Ø THEN RETURN
2040 PRINT @7,"SAVE MUSIC TO TAPE"
2050 PRINT:LINE INPUT "FILENAME:";A$
2060 OPEN "O",#-1,A$
2070 PRINT#-1,STR$(NN),STR$(TE)
2080 FOR I=1 TO NN
2090 PRINT#-1,N$(I)
2100 NEXT:CLOSE#-1:RETURN
```

Tandy users should change POKE 329 to POKE 282 in Lines 25Ø, 48Ø, 81Ø, 94Ø, 138Ø, 176Ø and 193Ø. Also change EXEC 41194 to EXEC 36Ø38 in Line 155Ø, and EXEC 36Ø55 to EXEC 46481 in Line 163Ø.



ESCAPE: A NEW ADVENTURE GAME

Start chipping at your chains in the evil king's medieval Alcatraz, and escape with the sacred amulet. Enter part one of *INPUT*'s new adventure game

This adventure game comes in several parts. You'll find it challenging to play, but despite the apparent complexity of the program it is still based on the simple adventure framework starting on page 264, and the text compressor, starting on page 428.

The main part of the program is written in BASIC, but unlike other adventures you may have typed in, there are no clues in the program as to how to solve the adventure. This means that even the programmer can enjoy playing the game. All the text is encoded using the text compressor, allowing a larger, more text-heavy adventure than would be possible on domestic machines with limited memories, as well as concealing any clues. The encoded text will appear in a later part.

Because of memory limitations, there is no version for the Vic 20 or Spectrum 16K. And as the Acorn program uses MODE 7 in order to fit into the available memory, it will not run on the Electron.

THE GAME

The sacred Amulet of the Nitpu has resided safely in your village for thousands of years. But recently, a disaster has happened—an evil king has stolen it. The amulet now sits somewhere in his forbidding, mist-shrouded castle, from which the only sound is the screaming of tortured prisoners.

There is no way into the fortress, so you have contrived a desperate plan to recover the amulet. To gain entry, you have allowed yourself to be captured by the tyrant's men. When you have the complete adventure you must explore the castle, find the amulet, and then escape. Beware of some of the characters you'll encounter on your quest!

ENTERING THE PROGRAM

The game consists of three sections: the BASIC program, the coded text and the decoder. Start entering the BASIC from this issue, but don't forget to SAVE the program on tape. The remainder of the BASIC is given in following parts, then you will get a numeric listing for the coded text. The decoder is the routine which you have already seen on pages 648 to 655. There will be instructions on putting the program together as you go, and when you have the complete program you can RUN it and show the evil king a clean pair of heels.

Like most full-scale adventures, Escape contains a great deal of programming—the completed game fills the BBC completely—and will require a considerable amount of keying in, particularly the coded text which occupies two parts of Games Programming. The effort won't be in vain, though, as you will see no clues if you list the game.

```
10 BORDER 7: PAPER 7: INK 0: CLEAR 64580
110 CLS : POKE 23658,8
120 LET NN=1: GOSUB 3960
130 PRINT "Remember to press (ENTER) after
   each instruction."
140 PRINT "PRESS (R) TO RESTART FROM
   SAVED POSITION."
150 PRINT "PRESS ANY OTHER KEY TO
   START."
155 LET D$=INKEY$: IF D$="" THEN
   GOTO 155
156 LET L=18: PRINT "HANG ON A
   MINUTE."
160 DIM O$(21,17): DIM K(21): DIM
   E$(21,17): DIM F(21): DIM R$(32,17):
   DIM R(32)
165 DIM L(22): RESTORE 4100: FOR Z=1 TO
   22: READ L(Z): NEXT Z
170 IF D$="R" THEN GOSUB 3870: GOTO
   270
270 CLS
280 IF L=-3 THEN GOSUB 1760
290 IF L<10 THEN GOSUB L(L): GOTO 330
300 IF L<23 THEN GOSUB L(L)
310 IF L=22 THEN LET L=16: GOTO 300
320 IF E$(L,1 TO LEN M$)=M$ THEN LET
   NN=61: GOSUB 3960: GOTO 490
330 IF K(1)=1 AND L=1 THEN LET
   NN=44: GOSUB 3960
340 IF K(2)=2 AND L=2 THEN LET
   NN=45: GOSUB 3960
350 IF K(3)=3 AND L=3 THEN LET
   NN=46: GOSUB 3960
360 IF K(8)=8 AND L=8 THEN LET
   NN=47: GOSUB 3960
370 IF K(7)=L THEN LET NN=48: GOSUB
   3960
380 IF K(9)=9 AND L=9 THEN LET
   NN=49: GOSUB 3960
390 IF K(10)=10 AND L=10 THEN LET
   NN=50: GOSUB 3960
400 IF K(11)=11 AND L=11 THEN LET
   NN=51: GOSUB 3960
410 IF K(12)=12 AND L=12 THEN LET
   NN=52: GOSUB 3960
420 IF K(14)=14 AND L=14 THEN LET
   NN=53: GOSUB 3960
430 IF K(15)=15 AND L=15 THEN LET
   NN=54: GOSUB 3960
440 IF K(17)=17 AND L=17 THEN LET
   NN=55: GOSUB 3960
450 IF K(19)=19 AND L=19 THEN LET
   NN=56: GOSUB 3960
460 IF K(20)=20 AND L=20 THEN LET
   NN=57: GOSUB 3960
470 IF K(21)=21 AND L=21 THEN LET
   NN=58: GOSUB 3960
480 IF K(6)=8 AND L=8 THEN LET
   NN=59: GOSUB 3960
490 FOR Z=1 TO 21
500 IF Z=6 AND K(6)=8 THEN GOTO 520
510 IF K(Z)=L AND Z<>L AND Z<>7
   THEN PRINT "THE□";O$(Z);"□IS HERE."
520 NEXT Z
700 IF I=1 THEN LET N$=V$(1)
710 IF I=2 THEN GOSUB 1970: GOTO 270
720 IF I=3 THEN GOSUB 3300: GOTO 270
730 IF I=4 AND L=3 THEN GOSUB 1760
740 IF I=5 THEN GOSUB 2800: GOTO 270
750 IF I=6 THEN GOSUB 3390: GOTO 270
760 IF I=7 THEN GOSUB 3440: GOTO 270
770 IF I=8 THEN GOSUB 3560: GOTO 270
780 IF I=9 THEN GOSUB 2450: GOTO 270
790 IF I=10 THEN GOSUB 2090:
   GOTO 270
800 IF I=11 THEN GOSUB 3520:
   GOTO 270
810 IF I=12 THEN GOSUB 3730:
   GOTO 270
```

```
10 CLR:POKE 53280,0:POKE 53281,0
20 PRINT"♥"CHR$(5)TAB(255)TAB(245)
   "LOADING TEXT"
40 OPEN 1,8,0,"TEXT,S,W"
50 INPUT#1,N:DIM A%(N)
60 FOR L=1 TO N:INPUT#1,A$:A%(L)=
   VAL(A$):NEXT L
```





```
70 INPUT#1,T:DIMZ%(T):FOR L=1 TO T:
   INPUT#1,A$:Z%(L)=VAL (A$):NEXT L
110 CLOSE 1
120 Z$="123456789012345678901234567
   8901234567890":ZZ$=ZZ$+ZZ$
130 SYS 52976
140 DIM O$(21),K(21),E$(21),F(21),R$(32),
   R(32);LG$=CHR$ (153)
150 RESTORE:PRINT"♥π":TX=1:
   GOSUB 9900
160 PRINT TAB(244)"□EMEMBER TO
   PRESS <□□□◪▁◩> AFTER"
170 PRINTTAB(11)"EVERY INSTRUCTION.":
   PRINT
180 PRINT TAB(5)"□RESS <□> TO
   RESTART GAME FROM":PRINT TAB(12)
   "SAVED POSITION."
190 PRINT:PRINT TAB(6)"□RESS ANY
    OTHER KEY TO START"
200 GET G$:IF G$="" THEN 200
210 L=18:PRINT"♥"TAB(255)TAB(247)
   CHR$(14)"□LEASE WAIT A FEW
   MOMENTS"
220 DIM O$(21),K(21),E$(21),FF(21),R$(32),
   R(32)
230 IFG$="R" THEN GOSUB 4460:GOTO 340
240 PP=0:BB=1:V=10:DW=1:D$="":
   M$="":II$="":TT$="":X=0:Q=0:
   KK=0:QQ=0:OP=0:K=0
250 SS=1:C=0:M=0:XX=0J=INT
   (RND(1)*18)+1:G=INT(RND(1)*18)+1
260 TT=1:II=1:VV=0:F=0:KK=INT(RND
   (1)*21)+1:TX=70:GOSUB 9950: JM$=Z$
270 FOR NN=1 TO 21
280 READ K(NN),F(NN): TX= (NN*2+124):
   GOSUB 9950:O$(NN)= Z$: TX=(NN*
   2+125): GOSUB 9950
290 IF O$(NN)="□"THEN
   O$(NN)=""
300 NEXT NN
310 FOR MM=1 TO 32
320 READ R(MM): TX= 167+ MM:GOSUB
   9950: R$ (MM)=Z$
330 NEXT MM
340 PRINT"♥"
350 IF L=-3 THEN 1970
360 IFL<10THEN ON L GOTO 1510,1560,
   1690,1840,1880,1930,1750,1420,1150:
   GOTO410
370 IFL<20THEN ON L-10 GOTO 1190,
   1260,1240,1330,1470,1640,1790,1340,
   1060,1310
380 IF L>22 THEN 390
385 ON L-9 GOSUB 1190,1260,1240,1330,
   1470,1640,1790,1340,1060,1310,1360,
   1140,2080
390 IF L=22 THEN L=16:GOTO 380
400 IF E$(L)=JM$ THEN TX=61:
   GOSUB 9900: GOTO 570
410 IFK(1)=1ANDL=1 THEN TX=(44):
   GOSUB 9900
```

```
420 IFK(2)=2ANDL=2 THEN TX=(45):
    GOSUB 9900
430 IFK(3)=3ANDL=3 THEN TX=(46):
    GOSUB 9900
440 IFK(8)=8ANDL=8 THEN TX=(47):
    GOSUB 9900
450 IFK(7)=(L) THEN TX=(48):
    GOSUB 9900
460 IFK(9)=9ANDL=9 THEN TX=(49):
    GOSUB 9900
470 IFK(10)=10ANDL=10 THEN TX=(50):
    GOSUB 9900
480 IFK(11)=11ANDL=11 THEN TX=(51):
    GOSUB 9900
490 IFK(12)=12ANDL=12 THEN TX=(52):
    GOSUB 9900
500 IFK(14)=13ANDL=13 THEN TX=(53):
    GOSUB 9900
510 IFK(15)=14ANDL=14 THEN TX=(54):
    GOSUB 9900
520 IFK(17)=15ANDL=15 THEN TX=(55):
    GOSUB 9900
530 IFK(19)=16ANDL=16 THEN TX=(56):
    GOSUB 9900
540 IFK(20)=17ANDL=17 THEN TX=(57):
    GOSUB 9900
550 IFK(21)=18ANDL=18 THEN TX=(58):
    GOSUB 9900
560 IFK(6)=8ANDL=8 THEN TX=(59):
    GOSUB 9900
570 FOR CC=1 TO 21
580 IF CC=6 AND K(6)=8
    THEN 600
590 IF K(CC)=L AND CC<>L AND
    CC<>7 THEN PRINT"THE□"O$(CC)
    "□IS HERE."
600 NEXT CC
610 GC=FRE(0):PRINT CHR$(14)"◘HAT
    NOW?":INPUT I$
620 IF I$=TT$ AND TT=0 OR I$=II$ AND
    II=0 THEN GOSUB 3560:GOTO 340
630 FOR SC=1 TO LEN(I$)-1
640 IF MID$(I$,SC,1)="□" THEN I=SC:
    GOTO 660
650 NEXT SC
660 IF I=0 THEN V$=I$: GOTO 670
665 IF (I=1)<1 THEN V$="": GOTO670
666 V$=LEFT$(I$,I-1)
670 T$=MID$(I$,I+1)
680 IF V$="GO"THEN V$=T$
690 AC$="":FOR CC=1 TO LEN(T$):AC=
    ASC(MID$(T$,CC,1)+CHR$(0))
700 IF AC<90 AND AC<>32 THEN
    AC=AC+32
710 AC$=AC$+CHR$(AC)
720 NEXT CC:T$=AC$
730 I=0
740 FOR H=1 TO 32
760 IF LEFT$ (R$(H),LEN (V$))=V$ THEN
    I=R(H)
770 NEXT H
```

```
10 MODE7:HIMEM=&7900
20 PRINT"LOADING TEXT AND
   DECODER":*L."DECODE"
30 z$=STRING$(255,"□"):z1$=STRING$
   (255,"□")
40 H=OPENIN "CODE"
50 DIMa%(204),z%(1165)
60 FORP=1TO204:INPUT#H,a%(P):NEXT
70 FORP=1TO1165:INPUT#H,z%(P):NEXT
80 CLOSE#H
90 CALLHIMEM+116
100 DSTRING=HIMEM+308
110 CLS:VDU23;8202;0;0;0;
120 PRINT''FNW(1)
130 PRINT"Remember to press <RETURN>
    after"'"each instruction":*FX202,
    48, 207
140 PRINT"PRESS <r> TO RESTART
    FROM SAVED POSITION"
150 PRINT"PRESS ANY OTHER KEY TO
    START":D$=GET$:L=18:CLS:PRINT
    "HANG ON A MINUTE"
160 DIMO$(21),K(21),E$(21),f(21),R$(32),
    R(32)
170 IF D$="r" THEN PROCQ:GOTO 270
180 p=0:b=1:V=10:dw=1:D$="":
    M$="":i$="":t$="":X=0:Q=0:
    k=0:qq=0:OP=0
190 s=1:C=0:M=0:x=0:J=RND(18):
    G=RND(18):t=1:i=1:v=0:F=0:
    KK=RND(21):JM$=FNX(FNW(70))
200 FOR n=1 TO 21
210 READ K(n),f(n):O$(n)=FNX(FNW
    (n*2+124)):E$(n)=FNX(FNW
    (n*2+125)):IFE$(n)="□"THEN
    E$(n)=""
220 IFO$(n)="□" THENO$(n)=""
230 NEXT
240 FOR m=1 TO 32
250 READ R(m):R$(m)=FNX(FNW(167+m))
260 NEXT
270 CLS
280 IF L=-3 THEN PROCA
290 IF L<10 THEN ON L GOSUB 1330,1380,
    1490,1640,1680,1730,1550,1240,990:
    GOTO 330
300 IF L<23 THEN ON L-9 GOSUB 1020,
    1080,1060,1150,1290,1440,1590,1160,
    940,1130,1180,980,1890
310 IF L=22 THEN L=16:GOTO 300
320 IF E$(L)=JM$ THEN PRINTFNW(61):
    GOTO 490
330 IFK(1)=1ANDL=1PRINTFNW(44)
340 IFK(2)=2ANDL=2PRINTFNW(45)
350 IFK(3)=3ANDL=3PRINTFNW(46)
360 IFK(8)=8ANDL=8PRINTFNW(47)
370 IFK(7)=(L)PRINTFNW(48)
380 IFK(9)=9ANDL=9PRINTFNW(49)
390 IFK(10)=10ANDL=10PRINTFNW(50)
```



```
400 IFK(11)=11ANDL=11PRINTFNW(51)
410 IFK(12)=12ANDL=12PRINTFNW(52)
420 IFK(14)=14ANDL=14PRINTFNW(53)
430 IFK(15)=15ANDL=15PRINTFNW(54)
440 IFK(17)=17ANDL=17PRINTFNW(55)
450 IFK(19)=19ANDL=19PRINTFNW(56)
460 IFK(2Ø)=2ØANDL=2ØPRINTFNW(57)
470 IFK(21)=21ANDL=21PRINTFNW(58)
480 IFK(6)=8ANDL=8PRINTFNW(59)
490 FOR c=1 TO 21
500 IF c=6 AND K(6)=8 THEN 520
510 IF K(c)=L AND c<>L AND c<>7
    THEN PRINT“The□”O$(c)“□is here.”
520 NEXT
530 INPUT“What now?”I$
```

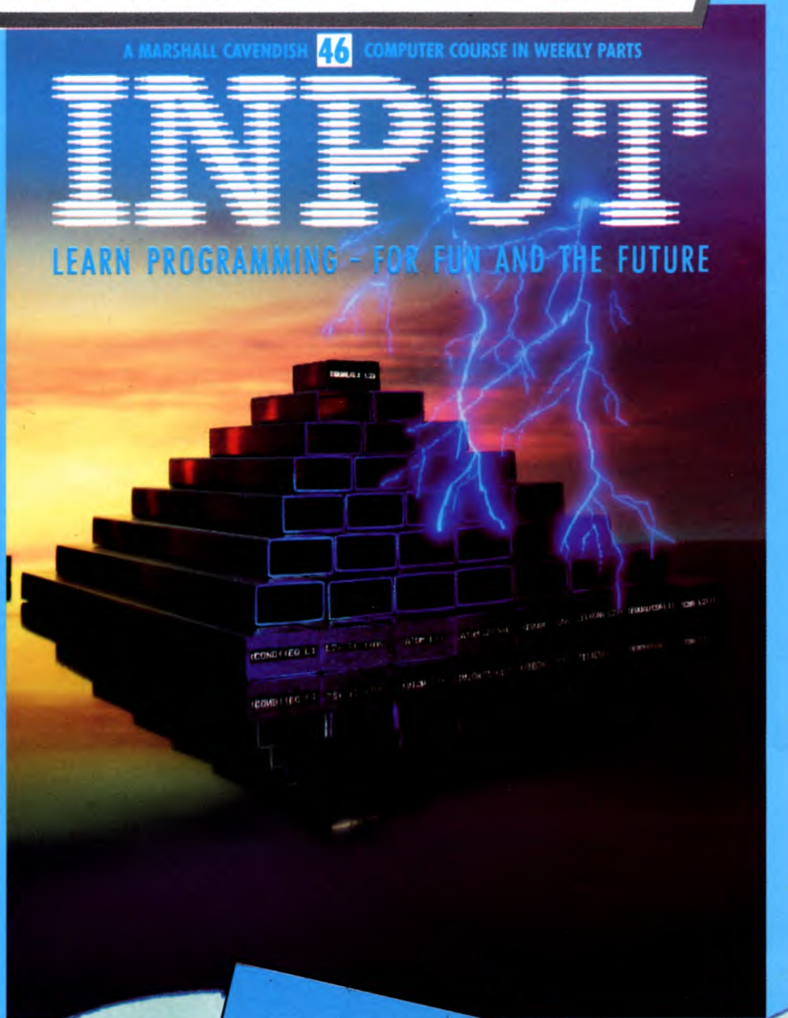
```
1Ø CLEAR 2ØØØ,32ØØØ:PCLEAR 5
2Ø CLS:PRINT“LOADING DECODER ...”:
   CLOADM “DECODE”
3Ø ZO$=STRING$(255,32):Z1$=STRING$
   (255,32)
4Ø PRINT“LOADING TEXT OFFSETS ...”:
   OPEN “I”,#-1,“”
5Ø DIM A(2Ø4):FOR P=1TO2Ø4:INPUT
   #-1,A(P):NEXT
6Ø CLOSE#-1
7Ø PRINT“LOADING COMPRESSED WORDS
   ...”
8Ø CLOADM “WORDS”
9Ø GOSUB 6ØØØ
1ØØ DIM O$(21),K(21),E$(21),F(21),R$(32),
    R(32)
11Ø CLS:RESTORE
12Ø WN=1:GOSUB51ØØ
13Ø PRINT“REMEMBER TO PRESS
    <ENTER> AFTER EACH
    INSTRUCTION”
14Ø PRINT“PRESS <R> TO RESTART
    FROM SAVED POSITION”
15Ø PRINT“PRESS ANY OTHER KEY TO
    START”
151 D$=INKEY$:IF D$=“” THEN 151
152 L=18:CLS:PRINT“HANG ON A
    MINUTE”
16Ø DIM O$(21),K(21),E$(21),F(21),R$(32),
    R(32)
17Ø IF D$=“R” THEN GOSUB 387Ø:GOTO
    27Ø
18Ø P7=Ø:B7=1:V=1Ø:DW=1:D$=“”:
    M$=“”:i7$=“”:T7$=“”:X=Ø:
    Q=Ø:K=Ø:QQ=Ø:OP=Ø
19Ø S7=1:C=Ø:M=Ø:X7=Ø:J=RND(18):
    G=RND(18):T7=1:I7=1:V7=Ø:F=Ø:
    KK=RND(21):WN=7Ø:GOSUB52ØØ:
    JM$=Z$
2ØØ FOR N7=1 TO 21
21Ø READ K(N7),F(N7):WN=(N7*2+124):
    GOSUB52ØØ:O$(N7)=Z$:WN=(N7*2+
    125):GOSUB52ØØ:E$(N7)=Z$:IF
    E$(N7)=“□” THEN E$(N7)=“”
22Ø IF O$(N7)=“□” THEN O$(N7)=“”
23Ø NEXT:K(7)=KK
24Ø FOR M7=1 TO 32
25Ø READ R(M7):WN=167+M7:GOSUB
    52ØØ:R$(M7)=Z$
26Ø NEXT
27Ø CLS
28Ø IF L=-3 THEN GOSUB 176Ø
29Ø IF L<1Ø THEN ON L GOSUB 133Ø,138Ø,
    149Ø,164Ø,168Ø,173Ø,155Ø,124Ø,99Ø:
    GOTO 33Ø
3ØØ IF L<23 THEN ON L-9 GOSUB 1Ø2Ø,
    1Ø8Ø,1Ø6Ø,115Ø,129Ø,144Ø,159Ø,116Ø,
    94Ø,113Ø,118Ø,98Ø,189Ø
31Ø IF L=22 THEN L=16:
    GOTO3ØØ
32Ø IF E$(L)=JM$ THEN WN=61:GOSUB
    51ØØ:GOTO49Ø
33Ø IF K(1)=1 AND L=1 THEN WN=44:
    GOSUB51ØØ
34Ø IF K(2)=2 AND L=2 THEN WN=45:
    GOSUB51ØØ
35Ø IF K(3)=3 AND L=3 THEN WN=46:
    GOSUB51ØØ
36Ø IF K(8)=8 AND L=8 THEN WN=47:
    GOSUB51ØØ
37Ø IF K(7)=(L)THEN WN=48:
    GOSUB51ØØ
38Ø IF K(9)=9 AND L=9 THEN WN=49:
    GOSUB51ØØ
39Ø IF K(1Ø)=1Ø AND L=1Ø THEN
    WN=5Ø:GOSUB51ØØ
4ØØ IF K(11)=11 AND L=11 THEN
    WN=51:GOSUB51ØØ
41Ø IF K(12)=12 AND L=12 THEN
    WN=52:GOSUB51ØØ
42Ø IF K(14)=14 AND L=14 THEN
    WN=53:GOSUB51ØØ
43Ø IF K(15)=15 AND L=15 THEN
    WN=54:GOSUB51ØØ
44Ø IF K(17)=17 AND L=17 THEN
    WN=55:GOSUB51ØØ
45Ø IF K(19)=19 AND L=19 THEN
    WN=56:GOSUB51ØØ
46Ø IF K(2Ø)=2Ø AND L=2Ø THEN
    WN=57:GOSUB51ØØ
47Ø IF K(21)=21 AND L=21 THEN
    WN=58:GOSUB51ØØ
48Ø IF K(6)=8 AND L=8 THEN WN=59:
    GOSUB51ØØ
49Ø FOR C7=1 TO 21
5ØØ IF C7=6 AND K(6)=8
    THEN 52Ø
223Ø WN=135:GOSUB52ØØ:IF H$=“Y”
    AND F(L)>1 AND E$(L)<>Z$ THEN
    PRINT“YOU CAN'T FIGHT THE ”;E$(L):
    PRINT“WITH BARE HANDS!”:GOSUB
    55ØØ:GOTO244Ø
224Ø IF H$<>IIY” AND W7=1
    THEN 244Ø
225Ø E7=RND(6):CLS
226Ø CLS 8
```



CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 46...



Stop wasting time muddling through
ose complex jobs by using your
omputer to PLAN THE BEST COURSE
pick your way through the maze of
lternatives

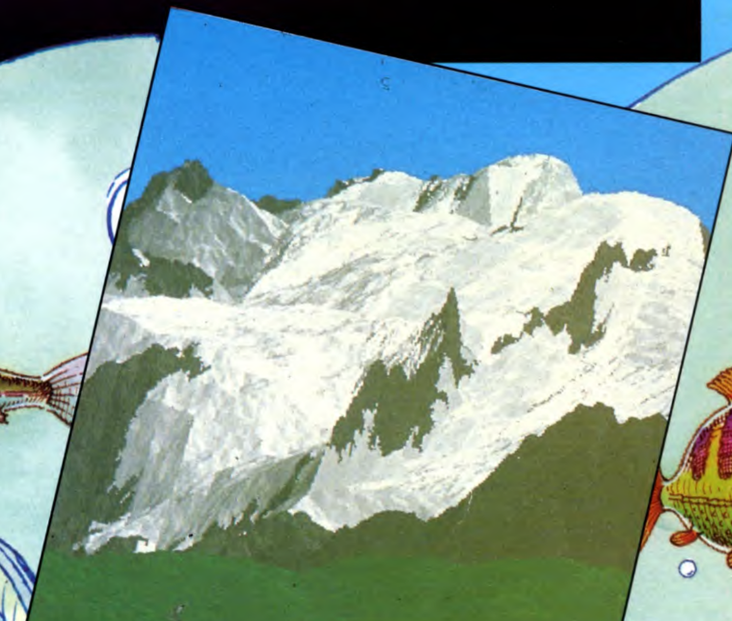
Banish silence from your
rogramming with interrupt-driven
USIC WHILE YOU WORK on the
OMMODORE and ACORN

Pretty patterns aren't the end of the
ory for FRACTALS. Use these
rograms to generate SHAPES FROM
ATURE

Willie's destined for more than six
ixels under in CLIFFHANGER, but it's
great undertaking—he'll live to picnic
nother day

Learn how to structure LISP programs
sing both inbuilt and user defined
nctions

. . . and for adventurous readers, part
o of ESCAPE



ASK YOUR NEWSAGENT FOR INPUT